游戏开发经典丛书

THOMSON

C++ for Game Programmers

C++游戏编程

(美) Noel Llopis 著
李鹏 贾传俊 译



- 如何高效地使用C++开发游戏
- 使用最为流行的技术
- 基于PC和主控台的开发技术
- 经过实践验证的众多思路和方法

清华大学出版社

游戏开发经典丛书

第一批书目：

《C++游戏编程》

《游戏音频编程》

《Java 2游戏编程》

《在线互动游戏开发》

《人工智能游戏编程真言I》

《游戏编程中的人工智能技术》

《3D游戏与计算机图形学中的数学方法》

http://www.thomsonlearningasia.com

ISBN 7-302-09048-3

9 787302 090489 >

定价：38.00元(附光盘1张)

组稿编辑：曾 刚 许存权

文稿编辑：鲁秀敏

封面设计：秦 铭

游戏开发经典丛书

C++游戏编程

（C++ for Game Programmers）

（美） Noel Llopis 著

李 鹏 贾传俊 译

清华大学出版社

北 京

内 容 简 介

本书从游戏开发的角度出发，把C++应用到游戏软件领域，介绍一些C++的实战经验，用常规的C++技术解决游戏开发者经常遇到的问题。重点讲述已经在实际的项目中应用的技术，而不是大段地罗列代码。

本书是游戏开发经典丛书系列之一，适合游戏开发人员、业余游戏软件开发爱好者和有关游戏软件开发培训班使用，也可以作为大专院校相关专业的参考书。

Noel Llopis

C++ for Game Programmers First Edition

EISBN：1-58450-227-4

Copyright © 2003 by CHARLES RIVER MEDIA,INC. a division of Thomson Learning

Original language published by Thomson Learning (a division of Thomson Learning Asia Pte Ltd). All Rights reserved.

本书原版由汤姆森学习出版集团出版。版权所有，盗印必究。

Tsinghua University Press is authorized by Thomson Learning to publish and distribute exclusively this Simplified Chinese edition. This edition is authorized for sale in the People's Republic of China only (excluding Hong Kong, Macao SAR and Taiwan). Unauthorized export of this edition is a violation of the Copyright Act. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

本中文简体字翻译版由汤姆森学习出版集团授权清华大学出版社独家出版发行。此版本仅限在中华人民共和国境内（不包括中国香港、澳门特别行政区及中国台湾地区）销售。未经授权的本书出口将被视为违反版权法的行为。未经出版者预先书面许可，不得以任何方式复制或发行本书的任何部分。

北京市版权局著作权合同登记号 图字：01-2003-2181

版权所有，翻印必究。举报电话：010-62782989 13901104297 13801310933

本书封面贴有清华大学出版社激光防伪标签，无标签者不得销售。

图书在版编目（CIP）数据

C++游戏编程/（美）罗比斯（Llopis,N.）著；李鹏，贾传俊译 北京：清华大学出版社，2004.9

（游戏开发经典丛书）

书名原文：C++ for Game Programmers

ISBN 7-302-09048-3

Ⅰ.C… Ⅱ.①罗… ②李… ③贾… Ⅲ.C语言 程序设计 Ⅳ.TP312

中国版本图书馆CIP数据核字（2004）第069967号

出 版 者：清华大学出版社 **地 址**：北京清华大学学研大厦
http://www.tup.com.cn **邮 编**：100084
社 总 机：010 62770175 **客户服务**：010-62776969
组稿编辑：曾 刚 许存权
文稿编辑：肖秀敏
封面设计：秦 铭
版式设计：张红英
印 刷 者：北京密云胶印厂
装 订 者：北京市密云县京文制本装订厂
发 行 者：新华书店总店北京发行所
开 本：185×260 **印张**：20.5 **字数**：468千字
版 次：2004年9月第1版 2004年9月第1次印刷
书 号：ISBN 7-302-09048-3 TP·6391
印 数：1～5000
定 价：38.00元（附光盘1张）

本书如存在文字不清、漏印以及缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770175-3103或(010)62795704

译 者 序

在国外，游戏软件开发已经成为一个产业，且已有近二十年的历史。而在中国则起步较晚。在软件行业不太景气的大背景下，游戏软件开发行业缓慢地发展着。可喜的是，近两年的网络游戏可谓风起云涌，忽然间，游戏开发成了很热门的行业。

国内一些大学纷纷开设游戏开发专业或者课程，社会上也有一些培训班，吸引着大批青年纷纷加入这个行业。但是游戏开发有着不同于一般软件开发的特点，这使得初学者或者刚入行者对此不太了解，而且市面上匮乏这样的参考书。清华大学出版社在这样的背景下，从国外引进了一套游戏开发丛书，本书是其中之一，主要讲述如何用 C++来开发游戏软件。

全书共分 3 个部分：

第 1 部分初识 C++威力。这一部分主要介绍 C++一些最基本的概念，如单继承、多重继承、常量及引用、C++模板以及异常处理。主要介绍了它们的性能开销情况。

第 2 部分性能和内存。这一部分主要介绍 C++的一些技巧和高级话题：如何避免性能方面易犯的错误，如何加速编程，如何有效地管理内存，怎样使用更容易、更有效的高级数据结构和算法。

第 3 部分专门技术。这一部分主要对实际游戏软件开发过程中用到的特定技术做一些介绍，同时讨论不同技术的优点。这些技术包括抽象接口、插件、运行期类型信息、对象的序列化、处理大型项目以及如何防止游戏崩溃。

参与本书翻译工作的有李鹏（北京邮电大学）、贾传俊（北京交通大学），王正盛、贾传俊做了审核，全书最后由黄丹卉和张慧做了校对工作。在本书的翻译和审校的过程中还得到了很多朋友的帮助和指导，在此一并表示感谢。

由于时间仓促，译者水平有限，书中若有错误和不妥之处，敬请广大读者不吝指教。

译者

2004 年 2 月

谨以此书献给我的父母，感谢他们的爱，以及这么多年来对我的鼓励。

也献给我的妻子Holly，感谢她为我所做的一切。

致　谢

首先，我应当感谢马萨诸塞大学阿默斯特分校的 Sandy Hill 教授。因为计算机图形学的有关知识就是他教给我的，他还促动我更深入地学习新知识。感谢他的课程，感谢他的指导。正是由于这些，我才最终选择了现在的职业道路，进而从事了我的游戏业职业生涯。

多谢 Ned Way，他不仅审阅了本书的部分底稿，帮我加了一些底层的细节，还四年如一日地陪伴着我一起工作，在这期间我从他身上学到了很多东西。

此外，需要感谢的人还有 Tom Whittaker、Kyle Wilson 和 Adrian Stone，感谢他们的评论，感谢他们关于 C++不同主题的一些有趣的讨论，本书就是在这些讨论后成形的。

也要感谢所有的游戏开发者，他们积极地与我分享信息，比如所著的图书，或者在杂志上撰写的文章、举办的讲座，或参加游戏开发的邮件列表（mailing list）。特别需要感谢 sweng-gamedev 邮件列表的朋友们那里的很多有趣的讨论，尽管有时我不能同意所有的观点。没有这种共享精神，我们可能只能停留在不断的重复学习之中。否则，我们就不可能取得如此快的进展。

更为重要的是，感谢我的妻子——Holly Ordway。没有她精神上的、实际的鼓励和支持，没有她对每一章毫无厌倦的校对，没有她提供那么多有用的建议，这本书是无论如何不可能完成的。还要感谢 Mipmap，我们家的小猫，她以自己的方式帮助着我：她趴在我膝盖上睡觉，于是我就不能站起来了，所以我不得不更快地完成了本书的撰写工作。

简　介

为什么写这本书

市面上已经有很多 C++的书了，几百本了吧！其中不乏这个领域（游戏开发）的非常好、非常经典的书，这些书在本书中也会提及。当然也有一些不是太好的书，这样的书则会很快被人忘记。那为何还要写一本这样的书呢？

几个月前，我读了一本很不错的书，是讲述刚发布的标准模板库（STL）的。我快速通读了这本书，它给我留下了很深刻的印象。所以我为之倾倒，我把它推荐给所有的同事，我觉得他们都应该读读这本好书。于是我想，这本书能不能更好一点呢？很意外，答案居然是 yes：这本书可以专门讲游戏编程。

那些 C++的巨著，都提供了很多不错的建议：它们能提高你的 C++编程能力，也能提高你设计出稳定的 C++程序的能力，但是它们的内容主要集中在 C++的通用知识。虽然也有一些讲述游戏编程的，但是不多。不仅如此，有些书甚至讲述了一些对游戏软件开发不利的建议，当你意识到这一点时，往往你已经陷得太深了，你的项目里大量使用了这些不合适的技术。

这就是我要写这本书的原因。它并不想取代那些 C++巨著。相反，是它们的补充。从游戏软件开发的角度出发，把 C++应用到这个领域。本书会介绍一些 C++的实战经验，指引你避开一些潜在的危险。本书还会讲述常规的 C++技术来解决一些游戏软件开发者不得不面对的特定问题。它会是你很好的帮手，帮你引向游戏软件开发之路，帮你迅速成为一名经验丰富的 C++游戏软件开发者。

当有疑问需要查证的时候，本书也能提供一些 C++开发的注重实效的入门材料。近来有一些书和文章详细说明了一些特定的技术细节，讲述的多是语言上的技巧。然而它们故意隐瞒了真相，几乎没有编译器支持这些技术。还有，它们也会使代码混乱。本书将重点讲述已经在实际的项目中应用的技术，还会鼓励读者弄清楚更多的试验结果与那些未经证实的方法。

游戏软件开发真的与一般的程序开发不同吗？你经常可以发现有些一般应用程序的一些需求和游戏软件的一样。但是在很大程度上，答案非常清楚：yes。下面所列的是游戏软件的一些特性：

- 可以交互运行；
- 保持一定的帧速率，一般最小每秒 30~60 帧；
- 接管机器的所有或者大部分；
- 大量地使用资源；
- 使用跨平台的开发方式。

举个例子吧。比如说，你为了检查电子邮件，按下了一个按钮，如果分配内存缓冲区的时间用了 50ms，那这样的响应时间是可以接受的。但是如果是在一个以每秒 60 帧运行的游戏中（每一帧用 16.7ms），这样的响应速度绝对是不能接受的。

这样的特性对于我们的游戏开发入门是一个限定。游戏开发是一个混合体，它有别于一般应用程序，它是实时的，它是与操作系统相关的。为了使游戏更有趣，需要开发数以 GB 计的游戏资源（图片、声效、音乐等）。还有，开发团队往往是由性格、能力差异很大的人组成的，开发的过程一定也很有意思。

游戏开发的 C++

前些年，也就是 20 世纪 90 年代，C 语言是游戏开发理所当然的选择。到了现在，C++取代了 C 语言的位置，变成了首选的语言。而且 C++确实很方便，由于 C 语言是 C++语言的一个子集，所以由 C 升级到 C++也是很容易的。

我们注意到，不是说一个程序是用 C++写的，它就是面向对象的了。有很多用 C++写的代码只是“更好的 C 版本”。而且，C++在一些平台上开发，还不是标准。这些平台有着严格的限制，比如手持设备和蜂窝电话。

从 C 语言到 C++语言的转化有两个主要的原因。第一个原因是由于 C++的复杂性。由于程序变得越来越复杂，需要寻找新的办法使得它们能够处理这种增加的复杂性。一句话，和 10 年前 C 取代汇编语言成为主流开发语言的原因一样，C++取代了 C 语言。

第二个原因是 C++成熟了。20 世纪的 90 年代，C++终于发展到了成熟稳定的阶段，C++标准也建立好了，编译器的支持也提高了。而且不同的平台上，甚至包括一些新的游戏机终端，也涌现出一些 C++的编译器。

由于这两个原因，再加上计算机运行速度越来越快，内存越来越大，C++就成了很好的选择了。然而，这并不是说，C++就是完美的了。差得远着呢！C++也有着自己的问题，有一些是由于它后向兼容 C 语言造成的，有一些是由于设计缺陷造成的。即使如此，C++有着这样那样的问题，它还是我们手边最好的工具。本书不想吹捧某种语言，它仅仅帮助读者在使用 C++开发游戏软件的时候更有效率一些。

虽然 C++有很多不好的方面，它的能力还是非常强大的。要想有效地使用它，还是很困难的，这也需要一个逐步的学习过程。然而与收获相比，还是值得的。C++是少数允许必要的时候访问计算机的底层的编程语言之一，它对系统编程来说是完美的，对游戏软件开发来说也是。诀窍在于知道如何避免纠缠于不必要的细节，而应该着眼于大局。C++编程有着平衡的技巧，同样的功能可以有很多个不同的实现方法，设计就是找到以下要素的平衡点：效率、可靠性、可维护性。

C++另外一个伟大之处在于，它是跨平台的。用同一套代码，根据游戏终端（指游戏机）的不同，可以在不同的平台上编译，和 PC 的情况一样，这是变得越来越流行的原因。有一些很不错的 C++编译器，像 STL 和其他一些库一样，可以适用于主要的游戏开发平台。可是，大多数底层的图像处理、用户交互、声音 API 还是与平台相关的，在不同的平台抽象出它们的细节是很有用的实践。

读　者

本书主要读者是使用C++开发游戏的软件工程师，或者相似的其他领域，比如实时图像处理或者系统开发。在这些领域有着C++丰富编程经验的老手，或者是有着编程背景的新手，都会从中获益匪浅。

为从本书获益更多，读者应当是使用C++有些年头，至少参加过一个完整的项目。读者应当熟习C++的机制，以便随时跳到后面的内容。如果非常了解C语言，想转到C++语言，还想转得很快，本书可以实现这样的愿望。配合其他一些C++基本概念的书，本书能缩短好几年的学习周期。

需要特别指出的是，你至少写过一些C++程序，熟悉它的语法，熟悉一些面向对象的基本概念，比如继承等。如果还不熟悉这些的话，那最好先放下本书。先找些第1章的末尾（见阅读建议）所列的C++书籍，花上几个星期，好好看看，然后再回过头来阅读本书。虽然这不是必需的，但熟悉了指针的概念、内存的分配、CPU寄存器，以及基本的计算机体系结构，会有助于读者从本书获益更多。

一些约定

会有一些特定的术语贯穿本书。这些术语大多数C++程序员都了解，但是缺乏准确的定义。然而，大多数情况下，它们的准确意思对于领会一些概念会很重要。不是在它出现的地方都解释，这里解释一下大多数主要的术语。

- 类：用户定义类型的一种抽象。类包含数据和建立在此数据之上的操作。
- 实例：内存的特定区域，保存有一个类的所有数据成员，一个类可以有很多个这样的实例。
- 对象：类的实例的另外一个叫法。对象就是通过实例化类创建的。

```
// This is an object of the class MyClass
MyClass object1;
// This is another one
MyClass object2;
// The pointer points to an object of the class MyClass
MyClass * pObj = new MyClass;
```

- 声明：源代码的一部分，用来说明一个或者多个名字。以下的代码就是对一个类的声明：

```
class MyClass {
public:
  MyClass();
  void MyFunction();
};
```

- 定义：代码的一部分，讲一个函数或者一个类的具体实现。以下的代码定义了函数 MyFunction：

```
void MyClass::MyFunction() {
    for ( int i = 0 ; i < 10 ; ++i ) {
    //  Do something...
    }
}
```

在每一章的内容里，有一些小的代码片断，从本书附带的光盘上可以找到完整的代码。我尽量让它们很简洁地表达它们要展示的概念，以免影响读者的阅读。你会看到所有变量和函数的命名采用了大小写混合的风格。另外，在变量命名的时候采用了基本的前缀，我认为这样可以增强变量的可读性，用不着害怕，它只是一个不完整的匈牙利表示法。

范围前缀

- m_：类的成员变量。可以从变量的命名上直接看出一个变量究竟是成员变量，还是一个栈里的局部变量，比如 m_HitPoints 就是一个类的成员变量。
- s_：类的静态成员变量，比如 s_HeapName。
- g_：全局变量。这种变量不会太多，比如 g_UserInfo 是一个全局变量。

类型前缀

- b：布尔变量。例如 bAlive。
- p：指针。例如 pItem。

范围前缀和类型前缀可以组合起来使用，比如说 m_bCanFly 和 s_pHeap 就是这样的例子。

本书主要讲述的是概念，不像有一些书那样罗列了大段的 C++代码，你甚至可以直接运用到自己的游戏项目里去，无须了解它究竟做了什么。相反，本书讲述的是这些概念（比如继承、虚函数等）怎么工作的比较细节的问题，它们会涉及哪些性能消耗，以及为了更好地使用它们，我们要遵从什么样的规则。

本书附带的光盘里有一些代码，提供这些代码的目的是举例说明书里所提到的概念。讲述构造函数并列举一点代码片断是一回事，真实地看到一个小程序怎么样应用构造函数是另外一回事。两方面读者都需要了解。

为了使代码更为整洁、清晰，而且能够准确表达其目的，我们特别让这些代码实现得很直观。因此这些代码通常缺少错误检测，或者缺乏层次上的提炼和抽象。这些代码你不能直接拿来应用到商业程序中去。

然而，还是可以使用它们的。首先修改，尝试着使用它们，最终改成适合你需要的代码。或者干脆重写它们，并运用从本书中学到的知识来指导你去开发一个游戏引擎。

目 录

第 1 部分 初识 C++威力

第 1 章 继承 2

1.1 类 2

1.2 继承 3

1.3 多态和虚函数 6

1.4 是否使用继承 8

1.5 使用或者避免使用继承的时机 10

1.6 继承的实现（高级话题） 11

1.7 性能分析（高级话题） 13

1.8 替代方案（高级话题） 15

1.9 程序架构和继承（高级话题） 16

1.10 结论 17

1.11 阅读建议 18

第 2 章 多重继承 19

2.1 使用多重继承 19

2.2 多重继承的问题 23

2.3 多态 26

2.4 什么时候使用，什么时候避免多重继承 27

2.5 多重继承的实现（高级话题） 28

2.6 性能分析（高级话题） 30

2.7 结论 32

2.8 阅读建议 33

第 3 章 常量及引用 34

3.1 常量 34

3.2 引用 41

3.3 强制转换 45

3.4 结论 49

3.5 阅读建议 49

第 4 章 模板 51

4.1 寻找通用代码 51

4.2 模板 56
4.3 使用模板的不足之处 60
4.4 使用模板的时机 61
4.5 模板专门化（高级话题） 62
4.6 结论 64
4.7 阅读建议 64

第5章 异常处理 66
5.1 错误的处理 66
5.2 异常的使用 70
5.3 异常的保护代码 76
5.4 异常的开销分析 81
5.5 异常的使用时机 82
5.6 结论 83
5.7 阅读建议 83

第2部分 性能和内存

第6章 性能 86
6.1 性能和优化 86
6.2 函数类型 89
6.3 函数内联 93
6.4 函数开销更多的方面 97
6.5 避免复制 101
6.6 构造函数和析构函数 106
6.7 数据缓存与内存对齐（高级话题） 110
6.8 结论 113
6.9 阅读建议 113

第7章 内存分配 115
7.1 栈 115
7.2 堆 116
7.3 静态分配 120
7.4 动态分配 122
7.5 定制内存管理 127
7.6 内存池 135
7.7 万一出现紧急情况（内存耗尽） 140
7.8 结论 141

7.9 阅读建议 141

第 8 章 标准模板库——容器 143

8.1 STL 概述 143
8.2 用还是不用 STL 145
8.3 序列式容器 147
8.4 关联式容器 156
8.5 容器适配器 165
8.6 结论 169
8.7 阅读建议 170

第 9 章 STL 算法及高级主题 171

9.1 算符（函数对象） 171
9.2 算法 174
9.3 字符串 182
9.4 分配算符（高级话题） 188
9.5 当 STL 不满足要求时（高级话题） 190
9.6 结论 191
9.7 阅读建议 192

第 3 部分 专门技术

第 10 章 抽象接口 194

10.1 抽象接口 194
10.2 通用 C++实现 195
10.3 作为绝缘层的抽象接口 197
10.4 作为类特征的抽象接口 201
10.5 其他方面 207
10.6 结论 208
10.7 阅读建议 208

第 11 章 插件 210

11.1 对插件的需要 210
11.2 插件结构 212
11.3 插件的组装 220
11.4 插件的应用 220
11.5 结论 222
11.6 阅读建议 222

第 12 章　运行期类型信息 223
12.1　不使用 RTTI 进行工作 223
12.2　使用 RTTI 224
12.3　标准 C++RTTI 226
12.4　自定义 RTTI 系统 230
12.5　结论 241
12.6　阅读建议 241
第 13 章　对象的创建与管理 242
13.1　对象的创建 242
13.2　对象工厂 244
13.3　共享对象 251
13.4　结论 261
13.5　阅读建议 262
第 14 章　对象的序列化 263
14.1　游戏实体序列化概述 263
14.2　游戏实体序列化的实现 266
14.3　组装起来 274
14.4　结论 275
14.5　阅读建议 276
第 15 章　处理大型项目 277
15.1　逻辑结构与物理结构 277
15.2　类和文件 278
15.3　头文件 279
15.4　库 290
15.5　配置 293
15.6　结论 294
15.7　阅读建议 294
第 16 章　防止游戏崩溃 296
16.1　使用断言 296
16.2　刷新机器状态 302
16.3　处理“坏”数据 304
16.4　结论 308
16.5　阅读建议 309
关于附带光盘 310

第1部分　初识C++威力



当你请教你喜爱的篮球运动员成功的秘诀是什么时，他肯定会说，机会，是机会。而机会完全来自于扎实的基础。他把大量的时间花在一遍又一遍的训练上，一年又一年地重复着左手运球、右手运球、还有扣篮。在掌握了这些基本的运球和一些技巧之后，他才可以在运动场上展示他那有效的攻防和惊人的球技。

从某种意义上说，C++和那些基于技巧型很强的运动非常相似。开发出大型的、让人吃惊的程序是可能的，但这需要很扎实的功底，这功底源于对C++的掌握和深刻理解。成熟的C++游戏往往让玩家敬畏于开发者的技术，它们每秒可以达到60帧的平滑动画，能充分利用硬件的性能。在开发这样的游戏之前，还是先学学C++吧，学习它的方方面面。

在这一部分将介绍C++不同于C的新特性。将看到它们是怎样使用的，它们内部是如何实现的。最重要的是，它们的性能是怎样的：了解什么时候不用这些特性和什么时候使用它们一样，都很重要。

第1章 继 承

本章将讲述：

- 类
- 继承
- 多态和虚函数
- 是否使用继承
- 使用或者避免使用继承的时机
- 继承的实现（高级话题）
- 性能分析（高级话题）
- 替代方案（高级话题）
- 程序架构和继承（高级话题）

继承是高级C++话题里经常谈及的基本概念，本书也会多次谈到它。继承允许在已经存在的类的基础上创建新的类。理解它是怎样工作的，怎么使用，以及使用它会有怎样的结果，这些都是为了用C++进行游戏开发的重要步骤。

1.1 类

对一个初学者而言，C和C++主要的区别在于类的概念。对一个了解C++不多但有着经验的C程序员来说，C++不过是“带类的C”而已。然而，对类来说，它们不仅仅是句法的美化。

从小的方面来说，类是把数据和函数关联起来的一种方式。对象是一个类的特定实例，每一个对象都有自己的数据，但是同一个类的所有对象共享同样的函数。

类的数据部分和C的结构没有什么区别。惟一的是C++提供了3个级别的访问方式：公有、保护、私有。默认的级别是私有，C的结构的数据成员都是公有（public）的。下面的数据结构是一样的，注意为了允许在外部访问成员变量，需要定义它们为public类型。

```
struct Point3d
{
   float x;
   float y;
   float z;
```

```
};

class Poinc3d
{
public:
   float x;
   float y;
   float z;
};
```

后面将会看到，不同平台上的编译器差异是很大的，这主要是因为它们的代码生成规则不一样。

另外一个不同的方面是编译器优化处理的不同。尽管如此，上面所说的结构和类都会生成完全相同的汇编代码，而且在不同平台上运行时，会有同样的性能特性。

可是把函数和数据关联起来，有什么好处呢？这就是句法美化给类带来了个性，它使得我们可以清晰地表示我们想对一系列数据进行的操作，比如为了得到一个向量的长度，使用 objToCamera.Length()比 Length(objToCamera)更为自然一些。

但这仅是一个错觉而已。从内部的情况来说，C++编译器可以把成员函数的调用改为一般的 C 函数调用，不同之处在于函数的第一个参数是一个代表要执行该函数的那个对象的指针。接下来的代码说明了成员函数调用及编译器的内部解析。

```
Vector3d objToCamera = object.GetPos() - camera .GetPos();

// This function call
float fDist = objToCamera.Length();

// Would be transformed to something like this
float fDist = Vector3d_Length(&objToCamera);
```

这并不是说类不重要。它是 C++其他一些特性的基础，正是这些特性，我们可以用面向对象这种更自然的方式编程。用这种方式编程，让人感到自然，让人感到愉快。为了真正看到 C++的威力来自何处，必须了解继承。

1.2 继 承

继承使我们可以很容易地在已有类的基础上创建一个与之不同的类。这只要很少的工作，而且不必改动原有类。

继承使概念的描述更为直观。例如，创建了一个类代表游戏中一个角色：敌人（Enemy），

这个类关注于角色在屏幕上的动画，跟踪它击中了何处，人工智能 AI 的运行等。

```
class Enemy
{
public:
   void SelectAnimation();
   void RunAI();
// Many more functions
   private:
         int m_nHitPoints;
   // Many more member variable here
};
```

当到了关底，需要加一个新的角色老板（Boss）的时候，必须面对的一个决定就是想重用 Enemy 这个类的很多函数（比如跟踪它击中了何处，动画特性等）。但是老板这类角色还有着一般敌人角色没有的特性，可以把 Enemy 这个类的一些成员函数拿出来，放到一个新写的 Boss 类里，不幸的是，这样意味着刚才创建的那个类 Enemy 的代码的封装性遭到破坏，而且这将导致多处维护代码的弊端。

这就是继承大显身手的时候了。可以创建一个新的类 Boss，它是从 Enemy 这个类继承来的。这意味着它将接受 Enemy 这个类的所有成员函数。此外，可以重载特定的部分以赋予 boss 特有的个性。这个例子中，可以重载 AI 部分，让老板（Boss）做与一般敌人（Enemy）完全不同的事情，让它表现出与众不同的特性。下面就是使用继承的 Boss 类的大概样子：

```
class Boss : public Enemy
{
public:
   void RunAI();
};
```

这种情况下，Enemy 叫做父类，Boss 叫做子类。因为 Boss 是从 Enemy 继承而来。现在可以这样同时使用 Enemy 类和 Boss 类：

```
Enemy enemy1;
Boss boss;

//Do other things for this frame
enemy.RunAI();
boss.RunAI();
```

一个类的公有部分，公有变量和公有成员函数，是暴露给所有使用这个类的人的。类的保护部分只能被自己和自己派生出的类使用。最后，类的私有部分只能被本类使用，自己派生出的类也不能使用。如果对这些还不太熟悉的话，可参阅本章最后的 C++参考书里关于这几个关键词的详细语法解释。

有的时候，我们不想完全重载一个父类的成员函数，只是想加些新的功能。比如，想让 Boss 类和 Enemy 类表现一样，但是想在这之前加上额外的 AI 运算。RunAI()函数的实现大概是这样的：

```
void Boss::RunAI()
  {
  public:
        //First run the generic AI of an enemy
        Enemy::RunAI();
        //Now do the real boss AI on top of that
        //…
  }
```

从以上的代码可以看出，为了调用父类的成员函数，也就是要重载的成员函数，需要在调用前加上前缀，两个冒号。这是 C++的范围操作符，它指定了特定函数，变量或者类的范围（全局或者某个类）。这个例子中，想调用父类 Enemy 中的 RunAI()函数，而不是当前 Boss 类的 RunAI()函数。

这不会阻碍我们继承一个子类，从而派生出一个新的子类。比如，在游戏的关底，想创建一个完全与众不同的 Boss，新建一个类，叫 SuperDuperBoss。一个被继承的类不会限定只能从直接父类中重载函数，它可以重载所有的父类的公有和保护成员函数。这个例子将重载 Enemy 父类的另外一个函数。

```
        Class SuperDuperBoss : public Boss
{
        public:
              void RunAI();
};
```

如果想描述类之间的继承关系，这样会很繁琐，描述的句子里将总是反复出现父类、子类、派生等，让人不好理解。这和描述家庭关系相似，比如描述一个远房亲戚：这是我异父妹妹的远房的表弟，就不够直观。和一个家庭的族谱类似，类的图表可以非常简洁地提供关于类的信息。图 1.1 展示了迄今为止所创建的 3 个类的关系。

在这本书里将沿用如图 1.1 所示那样的图，也可以在其他的 C++文献中看到这种图。

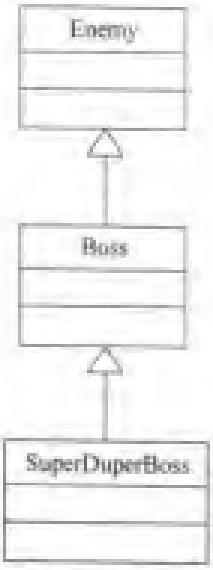


图 1.1　3 个 Enemy 类的关系图

1.3　多态和虚函数

到目前为止，所举的继承的例子是很简单的，但是它们很有实用价值。然而，如果打算像描述的那样，在实际的项目中使用继承，就会发现这样会很受限制，也很麻烦。主要的问题是，一个指向某个对象的指针，有着和被指的对象同样的类型，这听起来很奇怪吧。毕竟，一个整数永远是整型的，没有人会抱怨不可以用一个浮点指针（浮点数）去引用它。

还是上一部分的例子，假如有 20 个不同类型的敌人和 5 个不同类型的老板，而且本关里需要为每一个敌人调用 ExecuteFrame()函数一次，则需要跟踪每一个敌人的类型。这意味着需要有一个链表 list 保存每一种类型的敌人和老板，然后反复遍历这个链表。每当增加一个新的类型时，都得记住：增加一个链表，然后遍历它，这样做显然太啰嗦了。有没有一种方法，只需要用一个链表来保存敌人的种类，对每一个敌人调用 ExecuteFrame()函数，而不用管这个敌人究竟是什么类型的。

这正是多态要给我们的好处：多态使得我们可以通过父类的一个引用或者一个指针来指明一个对象，这听起来很悬乎，但是确实是这样的。请再读一遍，是不是有点感觉了？这是一个基本的概念，理解它非常重要。因为这样才能把本章和本书的余下部分读下去，这和你做的其他的关于 C++的功课一样。这里是一个简单的例子，说明多态可以让我们做什么：

```
class A;
//...
};
```

```
class B : public A{
//…
};

//We create an object of type B
B *pB = new B;
//But now we use a pointer of type A to refer to it
A * pA = pB;
```

尽管这个例子没有体现所有的深刻的东西，但多态确实是 C++的一个强大的特性，它允许忘记我们要操纵的对象的真实种类。例如，可以有一个函数，enemy 作为一个参数，该函数用来判断是否可以向这个 enemy 射击。这个函数大概是如下样子：

```
bool CanShootAt ( const Enemy & enemy )  const;
```

不要担心，如果你不了解 const 这个关键词，会在第 3 章里讲解它。一旦把一个老板 boss 加进游戏里，就需要确定是否可以向它射击。没有多态，我们要么被迫书写相似的函数——用某种 Boss 的对象作为参数，要么做一些很危险的事情，像传递一个 void 类型的指针，和一个表明它是什么类型变量的一个标记。如果增加了新的 enemy 类型，事情将变得更糟。多态对我们很有用，它允许我们只要一个函数，引用一个敌人类，不管敌人的具体类型是什么。这也是后面要用到的插件技术的机制，也可以用它扩展游戏的功能而无需重新编译（对程序的补丁或者用户创建 MOD 非常有用）。

回到前面的例子，利用多态的好处，程序会简化很多，保存所有的敌人在一个数组里，不管它是一般的敌人、老板抑或是终极老板，我们把它们同样对待。

但是这样有一个潜在的麻烦。记住为 Enemy 类和 Boss 类（从前者继承而来）编写了一个 RunAI()函数，考虑以下的代码段，它用到了多态：

```
Enemy * pEnemy = new Enemy;
pEnemy->RunAI();        // Enemy::RunAI() gets called
Enemy * pBoss = new Boss;
pBoss->RunAI();         //Which function gets called??
                // Boss::RunAI() or Enemy::RunAI()
```

当指针和对象都是 Enemy 类型时，则调用 RunAI()就是调用 Enemy::RunAI()。当指针是 Enemy 类型，对象是 Boss 类型时，调用 RunAI()会怎样呢？答案则取决于 RunAI()是不是虚函数（virtual function）。

一个函数说明为虚函数，表明在继承的类中重载这个函数时，当调用这个函数，应当查看以确定调用哪一个对象的这个函数。不然的话，就使用指针的类型作为判断的依据。

在例子中，想让老板使用 Boss 的 AI，想让每一个 enemy 使用他们自己类型的 AI。所以应当把 RunAI()函数说明为虚函数。下面是修订的 Enemy 类：

```
class Enemy
{
  public:
        void SelectAnimation();
        virtual void RunAI();
        // Many more functions
    private:
              int m_nHitPoints;
  // Many more member variables here
};
```

现在所有的 enemy 可以使用同样的代码，不管他们是不是老板。

```
enemy * enemies[256];
enemies[0] = new Enemy;
enemies[1] = new Enemy;
enemies[2] = new Boss;
enemies[3] = new FlyingEnemy;
enemies[4] = new FlyingEnemy;
// etc…

{
  //Inside the game loop
  for ( int i=0; i < nNumEnemies; ++i )
      enemies[i]->RunAI();
}
```

1.4 是否使用继承

当你手头上有一个锤子的时候，也许所有的事物在你眼里都是钉子。继承是一个非常强大的工具。一个很自然的反应就是：对所有的事情都想试试能否使用它。然而，与继承能解决的问题相比，不正确地使用它会带来更多的问题。应当考虑只在合适的场合下使用它，还有除此之外没有更简单的办法时。在没有很深入地了解面向对象设计的理论之前，这里有两个使用继承的准则。想了解更多的话，可参阅本章末尾的阅读建议。

规则 1　使用容器还是使用继承

通常，从一个既有类继承创建一个新类也就是建立了关系。如果类 B 从类 A 继承，也

就意味着，B 的对象也是 A 的对象。在以前的例子里，Boss 是 Enemy 类，所以遵循 Enemy 的规则。

想象一下，某人建议 Enemy 类从 Weapon 类继承，也就是说，敌人可以射击，可以做武器可以做的其他事情。

这很有吸引力，可是是否遵循规则呢？一个敌人是一个武器吗？显然不是。更准确的描述是一个敌人拥有一个武器。这叫做“容器”。它的意思是 Enemy 这个类应当拥有一个武器类型的成员变量，但是不应当从武器类继承。第一个准则就是：继承必须建立一个关系。

为什么费劲区分它们呢？即使 Enemy 类从 Weapon 类继承，程序可以编译，也可以正常运行。从短期的好处来说，可以节省打字的时间。当维护它或者打算修改它时，问题就会出现。如果敌人可以更换不同的武器怎么办？如果敌人可以同时拥有不同武器怎么办？如果敌人根本就没有武器怎么办？一个好的合乎逻辑的设计可以让这一切轻而易举。不正确的使用继承会让你的工作很难继续进行。

当在一个特定的条件下，还疑惑是否使用继承的时候，那就不要用。尤其是在一个比较大的项目的初期，还不了解继承的长远好处的时候。很多项目就是由于不恰当的使用继承造成项目流产，还不如用比较保守的方法可靠呢！

规则 2　行为和数据

当拥有了继承这个工具，还有上一个规则，一个有激情的程序员创建了一个新类，EnemyTough，它从 Enemy 继承而来，但它有两倍的 hit point（一个 enemy 受到打击，比如刀砍火烧之类后会失血，当血失完了，enemy 也就死了。hit point 就是用来衡量可以承受打击程度的一个参数）。这满足前一个规则，EnemyTough 确实是一个 Enemy，这还有错吗？

事情将变得奇怪，当查看 EnemyTough 类的代码时，这个类几乎是空的。惟一的内容是在构造函数里，在这个函数里指定比一般 Enemy 更多的 hit points。它看起来没用，实际上也没用！

EnemyTough 不应该是一个新类的原因是，它的不同主要在于数据，而不是对象的行为。如果 EnemyTough 不具有其他不同的行为，那它和一个通常的初始化为更多的 hit points 的 Enemy 对象没有什么不同。

所以第二个规则就是：继承一个新类是因为要改变它的行为，不是改变它的数据。

除了没做任何事（可以用很简单的办法完成）以外，为了改变数据而使用继承的一个很大的缺点就是会发生“组合爆炸”。也许有一个刀枪不入的敌人，如果不是在 Enemy 类里用一个标志来实现，可以创建一个新类 EnemyInvulnerable 来实现这一点。那 Boss 和 tough 更厉害，刀枪不入的 Boss 又怎么办呢？而且数据是可以在游戏的过程中自由改变的东西。也许敌人可以使用特别的能量变得可以在短时间内无敌，或者敌人可以获取另外的 hit points 而变得厉害。如果通过类而不是数据来实现的话，当游戏运行中需要变换 enemy 对象的类型时，事情将变得非常麻烦。

1.5 使用或者避免使用继承的时机

前面的部分解释了什么时候使用继承来实现类的创建是合适的。正常情况下，那就是你应当掌握的。大多数其他书籍在谈到这个部分时，也是这么说的。

还应当认识到，使用虚函数会有一些少许的性能损失，详情会在以后的章节中展开。1.6节会介绍继承和虚函数是如何实现的。就目前而言，如果使用虚函数，程序会运行得慢一点。

第一个要注意的问题是除非在十分必要的情况下，否则不应当使用虚函数。这听起来很浅显，可是有的时候就有人会去尝试，让每一个函数都是虚函数，仅仅是为了方便以后有人继承这个类，方便扩展这个类的功能。为了将来的扩充而做出这样的设计是很好的，但是在大多数情况下，最好别这么做。也许根本没有从这个类继承的对象，也许根本就没有人要重载这些函数。如果编译器足够聪明，它可以优化虚函数成为普通的函数，这时候虚函数机制就不必要了。但是C++语言不是这样设计的。所以目前来说，每调用一次虚函数，程序的性能就会有一点损失。

不要让每一个函数都是虚函数，除刚才上面说的原因之外，还有其他的原因。我们不能预测未来，所以除非意识到急切需要从一个特定类中继承，到那时，有人决定这么做，事情会变化太多，类需要重写。同时，创建一个新类，根本不想它将来的扩充性、私有函数、成员变量，这很容易，也很快。一旦把它们设为保护或者公共的虚函数，则开始担心如何使用它们。你要做的是把它们正确地分离开，只维护一个一致的状态。换句话说，除非现在有非常必要的原因，否则不要使用虚函数。

大多数情况下，这是我们担心的全部。在现行硬件条件下，调用虚函数的开支很小，所以几乎可以忘记这一点。

当它发生时，有时候我们支付不起某一个函数是虚函数。也许它是一个深藏于一个循环内部的函数调用，每一帧都会调用成千上万次。这是有可能的，如果这是一个很基本的函数，或者是一个内联函数（见第6章性能问题，那里会详细谈到内联函数）。在准备取代这个函数之前应当检查一下，看看它是不是虚函数，或者内联函数这个因素究竟是否损害了性能。如果是的话，应当把这个函数替换成一般函数，不要用虚函数。请记住在一个if语句里，单独的调用一般函数可能不会快多少（意思是代码要优化到关键点上去）。我们可能会考虑把循环移到一个类里面去，整个循环会以一个虚函数的形式被外部调用。这种办法只有一个虚函数调用，会比整个循环的开销小很多。

另外一个可能的性能损失是，当程序多次穿越虚边界时，这和上面说过的例子不一样，它不仅仅是一个被调用很多很多次的虚函数，而是一系列函数。比如，想象一下，一个图形渲染模块的虚接口，接口里的所有的函数都是虚函数。

```
class GraphicsRenderer {
public:
        virtual void SetRenderState(…);
        virtual void SetTextureState(…);
        virtual void SetLight (…);
        virtual void DrawTriangle(…);
//…
};
```

很不幸，GraphicsRenderer这个类的抽象不太合理。当想在屏幕上画一个三角形时将依次调用如下的虚函数：SetRenderState()、SetTextureState()、SetLights()以及DrawTriangle()。每一帧里，有多少个三角形就会调用多少次这样的调用。虚函数调用的开销会急剧上升。

这是一个极端的例子，这个程序画三角形的性能在现行的微机显卡上运行时，会把游戏跑死的。但是主要的开销来自于虚函数调用。那么怎样解决这个问题呢？可以把抽象移到更高层。不像刚才的接口，现在提出一个新的抽象接口，用它来画一个mesh（一些列三角形），代码大致如下：

```
class GraphicsRenderer {
public:
   virtual void SetMaterial(…);
   virtual void DrawMesh(…);
   //…
}
```

现在，画一系列三角形只要调用两个函数：SetMaterial()和 DrawMesh()。减少了虚函数的调用次数，从一个三角形4到5个函数调用，到即使成百上千也只有两个虚函数的调用。注意我们是怎样实现这一改变才是主要的，接口的这种根本改变有一个重要的结果，那就是所有调用它的代码，可以像一般的程序那样调用它。所以，当第一次设计一个接口的时候，一定要慎重考虑。

1.6 继承的实现（高级话题）

为了更进一步理解继承，学习继承究竟是怎样实现的就显得势在必行了。编译器的开发者对于如何实现继承有着很大的自由度。他们可以想怎么做就怎么做。幸运的是，对于每一个在微机上能碰到的编译器，在实现继承方面几乎是一样的，只有稍微的差别。

首先来看看一般的函数是怎么处理的。这很容易，因为一个特定的函数调用会映射到特定的代码，无论是编译阶段还是链接阶段，编译器都能计算出这个函数的地址。在程序

运行期，所有要做的工作就是调用那个固定的地址。

虚函数就有意思了，因为被调用的代码不仅依据调用的特定函数，还依据被调用的对象的种类。通常，这是通过虚函数表（vtables）来实现的。

vtables 也就是 virtual tables。我们经常称之为 vtables。vtables 就是一个表，它里面的每一个指针都代表一个函数。每一个类都有一个 vtables，只要它包含至少一个虚函数。而且每一个函数都会有一个索引代表函数在表中的位置。vtables 可以凭此来指明运行时调用哪个具体的函数。

还有一些关于 vtables 的重要的方面需要指出，vtables 仅包含虚函数的指针。一般的函数的地址还是编译时计算出来的，直接在代码里调用。这意味着，对于每一个虚函数，只是多花费一点字节数和一点性能上的损失，但是没有影响一般函数的性能。这也是 C++最初设计的准则之一：你无须关注你用不到的特性。

还有一个重要方面是，每个类只有一个 vtables，而不是每一个对象一个。这很重要，因为通常情况下，一个单独的类会有很多个对象的实例。举例来说，如果游戏中有一个地形，有一个类代表一个地形块。假如地图的大小是 256×256，就有 65536 个对象。幸运的是，只有一个 vtables（假设所有的地形块都是同一个类的实例，而这个类至少有一个虚函数）。

一个特定类的的所有对象拥有同样的 vtables。所以不是每一个对象都有自己的 vtables，每一个对象有一个指针，该指针指向那个类的 vtables。要不然，程序怎么在运行的时候能够知道某个对象是哪个类的实例？通常，该指针是对象的第一个入口。所以，一个对象的大小额外增加了一个指针的大小，假如这个对象有虚函数的话。现行平台上，一个指针多是 4 个字节。刚才地形的例子中，意味着将多使用 64KB 的空间来存储 vtables 指针，还要加上 vtables 自身的很少的一些字节数。与获得的好处相比，这点内存耗费算不了什么。但是应该认识到这确实是一个问题，尤其在一些内存有限制的平台上。

准确地说，不是每个对象都有 vtables 的指针。即使每一个对象所属的类继承于同一个类。只有拥有虚函数的类才有 vtables。这是很细微但是很重要的区别。意思是说，可以对一些小类或者基本类，比如 vector 向量、matrix 矩阵，只要没有虚函数，就可以自由使用继承。再说一遍，同样的规则仍然起作用：如果没有使用一个特性，就不要关注它的性能和内存消耗。

图 1.2 概略表明了一个带有虚函数的类的内存布局，虚函数实现的详细材料可参阅章末的阅读建议。

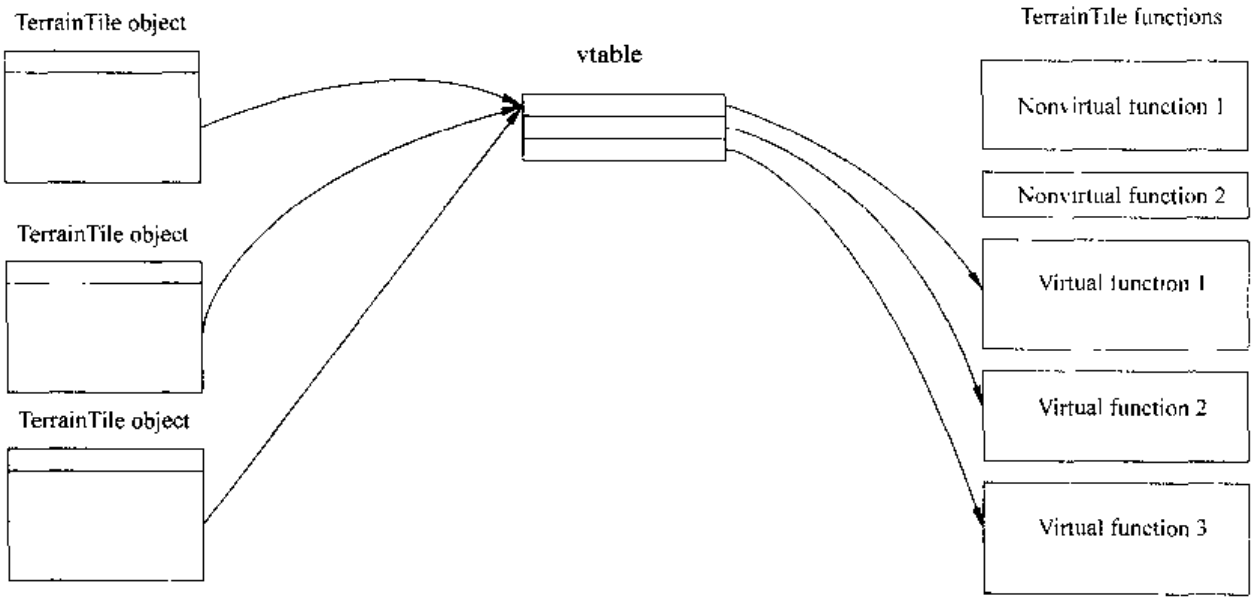


图 1.2 非虚函数、虚函数、类的 vtable 以及该类的一个对象的内存布局图

1.7 性能分析（高级话题）

本章自始至终，讨论了使用虚函数的性能损失。我们也曾说过，性能损失不会太大。但是性能损失不能完全忽略。损失到底有多大呢？每一次调用虚函数你是否都付出这种损失？每一个平台上是怎样不同的？下面是运行通过基类的指针来调用虚函数时，所发生的一切：

第 1 步：开始于一个调用：pEnemy->RunAI()。

第 2 步：取到对象的 vtable 的指针（如图 1.3 所示）。

第 3 步：从 vtable 那里获得入口的偏移量，相当于要调用的函数的指针（如图 1.4 所示）。

第 4 步：根据 vtable 的入口找到函数地址，并调用函数（如图 1.5 所示）。

第 4 步和一般的函数调用一样，所以额外的开销就是第 2 步和第 3 步。这两步使整个过程开销最大。可是具体是多大呢？不幸的是，这很难回答。

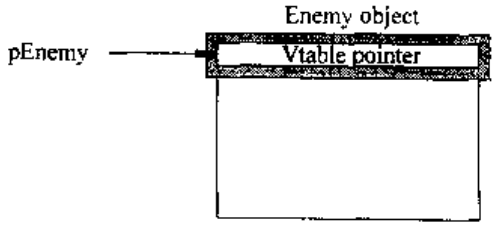


图 1.3 取到了对象的虚函数表的指针

不仅仅因为不同平台的开销差别很大，还有缓存（Cache）级别、管道线（pipelinc）的不同使得我们给不出一个很好的答案。

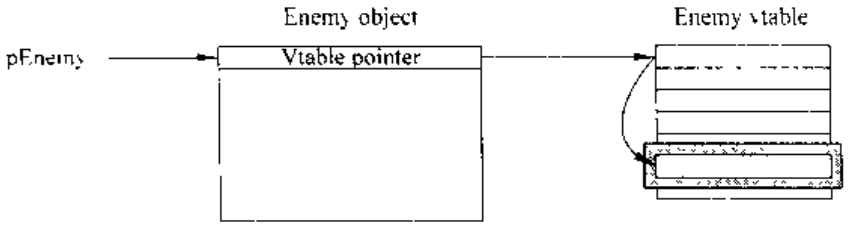


图 1.4　取到了函数的入口

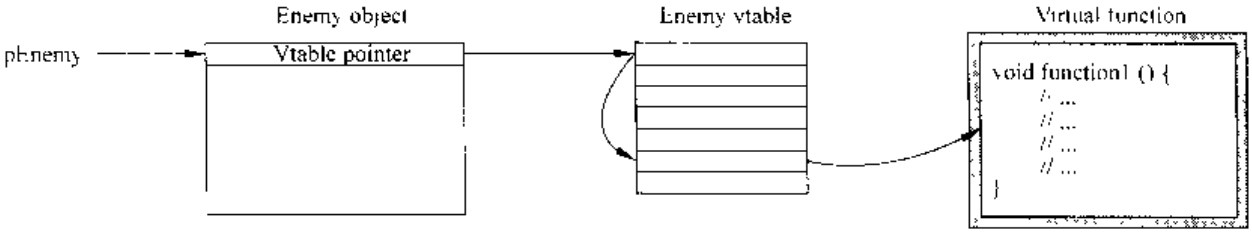


图 1.5　真正调用虚函数

可以忽略任何性能损失。先编写游戏代码，然后评估每一个函数，找出是否由于虚函数造成了瓶颈。即使如此，把优化放到最后总是一个不错的普遍做法，性能问题先放一放。首先，很难断定软件运行速度慢确实是虚函数造成的。一个循环的内部，可能由于算法设计的不合理而造成开销很大。虚函数对程序速度的的影响很难那么直观地表现出来。虚函数开销的后果是微妙的，它遍布在程序的各处。更为重要的是，即使想限制虚函数的开销，也许再改已经来不及了。如果整个游戏的架构就是基于继承和虚函数实现的，而想改变游戏的结构，去掉所有使游戏变慢的虚函数的调用，那就相当困难了。

最好的办法是，要对程序中由于虚函数的调用而带来的影响要有合理的认识。并且在写游戏的时候，牢记于心。然而，一旦游戏运行了，就不要再让那些虚函数的理论烦我们了。对程序做些整体的性能分析吧。

大多数时间，额外的开销是可以忽略的。一般情况下，程序慢不了多少（比如说，一般情况下瓶颈在于 CPU 都要等待 FPU 的处理完成）。有时候由于调用循环层数太深太频繁，我们能感受到性能明显降低。

最大的性能损失一般没有机会碰到，而不是第 2 步和第 3 步。虚函数通常不是内联函数，所以如果把一个虚函数定义为内联函数的话，将会导致显著的损失。应当避免把内联函数定义成为虚函数的想法。这可能意味着，把接口的设计弄到一个低水平的层次上。如果确实是这样的话，是该反思设计的时候了。

另外一个很大的问题是，vtable 在数据缓存中查找所带来的冲击。如果有很多不同的类，而且所有的类都是以随机的顺序执行的，每一帧我们需要停下以查找不同的类的 vtable 的入口。这将导致数据缓存返回的是其他数据，也有可能，数据缓存压根没有我们需要的 vtable 数据。在这种情况之下，或者那个平台根本就没有数据缓存或者很小，那虚函数带来的附加开销就会很明显，可能会极大地降低游戏的性能。

我们需要时刻关注由于虚函数调用而带来的附加开销吗？答案是没有必要。编译器的

开发人员有办法，可以让程序运行得尽可能地快。所以如果有什么便捷的方法可以让程序运行得快一点，他们会改进编译器的。

只要仔细检查 vtable 的跳转就可以了。这是大多数的情况，也是最通常的情况。通过对象直接调用一个虚函数，而不是通过对象指针或者引用调用它。这种情况之下，编译器会知道，会相应调用非虚函数。一个相似的情况是，如果使用的是一个指针或者一个引用，都是和对象一个类型的，编译器可以算出函数在 vtable 中的索引，从而直接调用那个函数。

```
Enemy   *   pEnemy1 = new Boss;
PEnemy1->RunAI();                  //We pay the virtual function cost

Boss * pEnemy2 = new Boss;
pEnemy2->RunAI();   //Virtual function call optimized out
```

1.8 替代方案（高级话题）

你已经了解继承的很多内容了，你仍然没有确信是否使用它。额外的内存消耗和程序的性能损失看起来对你影响很大。还有，你已经用 C 而不用继承，你的程序已经管理得很好了，为什么非要用继承呢？

这确实是个问题。最好的办法是在不用继承的 C 环境下，如何实现继承那样的功能。这样的办法有好几种。

- 一个结构和很多条件分支语句。可以创建一个 C 的结构，它包含了所有在继承类中可能用到的数据。专门添加一个表示类型的成员变量，在不同的应用场合，依据这个类型变量的不同取值去实现不同的功能。除了代码不够简洁，维护起来像恶梦之外，内存也会浪费很多，因为该结构的大小都一样（等于想使用的最多的数据的大小）。一旦程序里有不止一个的 if 语句，性能也会比使用继承的差。很多条件跳转语句的存在将导致程序性能下降很快。
- 一个结构，但是使用 switch 语句而不是 if 语句。这样源程序（代码）看起来就整洁多了，但是问题是编译器会生成和使用多个 if 语句情况一样的代码。这样会导致和前面同样的后果。在很极端的情况下，编译器会替换所有的 if 语句成一个跳转表。这种情况下，最好的可能是获得和使用虚函数同样的性能。不幸的是，通常没有可靠的方法强制编译器根据 switch 语句创建这么一个跳转表。
- 指针表。可以为每一个类型的对象创建一个函数指针表，然后在每一个结构里安排一个指针，该指针指向函数指针表的合适位置。我们需要特别小心地为每一个结构安排指针，以及正确地指向函数指针表。这样的代码可读性不是很好，调试也不是一件愉快的事情。这个办法是不是有点眼熟，当然了，就是又一遍实现了虚函数，但是没有得到继承的其他好处（可变的对象大小，私有、保护的成员等）。性能也会和使用虚函数和继承时一样，但是花费了更多的时间，做了更多的工作。

还是让编译器来做这些让人忙得团团转的事吧。

从中得出的结论是，如果需要继承的特性和虚函数，就直接用，而不是再实现一个。在优化代码方面，编译器会做得更好。

1.9 程序架构和继承（高级话题）

也许你回想起了，在前面的章节中，出现了所说的使用继承有一些潜在的危险等警示性的文字。这个问题远非小事，比如会带来不必要的性能损耗或者会带来额外的内存使用量的增加。实际的问题是，广泛的使用继承会给你的代码带来哪些问题？

类的目的是让你可以为一个抽象的概念建模，抽象出它们的具体实现。要想实现这个也很简单，只要提供最小的必要的公共接口，把所有复杂的私有函数和私有成员隐藏起来就可以了。记住，创建一个有着清晰的公共接口的类是一个挑战。你应当取舍，哪些应该暴露出来，哪些应该隐藏起来。要学会怎样在提供我们所需的灵活性的同时，接口还要最小。这不是一件容易的事，这项工作也不是那么直来直去的。

再把继承考虑进来，就好像一个军队在两个前线作战。不仅要关注这个类的当前使用者，还要注意将来的可扩展性，把将来需要暴露的设为保护类型。事情突然变得困难起来。

其他的使用的主要问题，继承会对程序的整体架构带来影响。使用继承过多会使继承构成复杂的链甚至构成复杂的继承树。也许一个继承树会有几十甚至上百个类，有的会有五六层或者更多的深度。虽然每一个类都是按照正确的原则从某类派生出的，事情仍然不可预知。拥有多层继承关系的这些类，会构成一颗巨大的继承树，这使得程序员想要理解代码究竟做什么的时候，就比较费劲了。查看函数调用时，需要对这颗树跟踪过来，跟踪过去，才能知道究竟是哪一个函数被调用了，这取决于特定的类重载了哪一个虚函数。

最糟糕的是，继承使得程序设计变得困难起来。一般的软件（游戏软件更是这样）都需要一定的灵活性。当情况发生变化，比如游戏发行公司提出了新的需求，或者为了增加软件的竞争力，需要增加一点新特性让软件更有意思。拥有一套能适应这种变化的代码库，是一个非常值得努力的目标。我们要做的最后的事情是两周内把游戏移植好，并且不能对程序做太大的变化。这看起来仅仅是琐碎的小事吗？不对，实际上不是这样的。

难道就没有别的办法了吗？前面提到过容器（包含关系）和继承是有区别的。容器模型“拥有”一个关系，而继承模型“是”一个关系。继承不应当应用到拥有某种关系的模型上。然而，有时候，当用继承比较合适时，还想用容器。

容器和继承不一样，容器不会加大程序设计的难度。事情仍然很有灵活性，有可扩展性。因为对象是包含在对象里的，没有必要知道是谁拥有它，它只需关注按照别人的吩咐，做自己的事情就可以了。在某种程度上来说，类越简单，使用起来就越容易，使用它来并发或者维护代码就越容易。

这并不是意味着用容器来取代所有的继承。只是当使用继承有问题的时候，才会用这种办法取代它。这样一种情形，类之间构成很深的继承链，考虑好在某些地方用容器，就

会打断这个继承链，这样，一个大的复杂的继承链就会变成 2 个或 3 个小的、简单一点的继承链。

如图 1.6 所示，这是一个这种变化的实例。一个拥有 5 层深度的继承链，用容器的方法做了些变形，最大的继承链才 3 层。注意继承的箭头和包含的箭头，它们是有区别的。

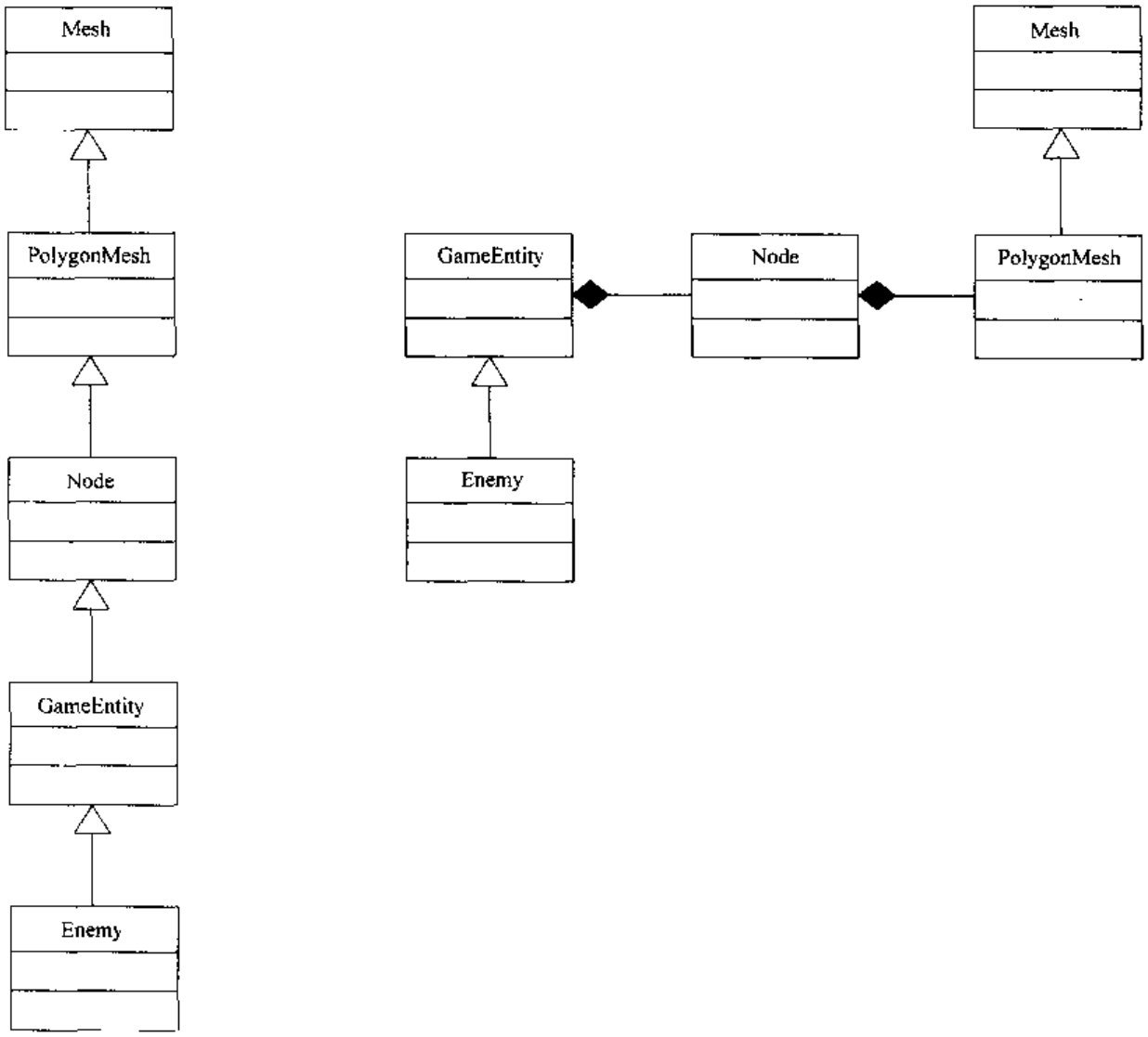


图 1.6 一个深度很高的继承链可以用容器的方法实现

1.10 结 论

在快速地回顾了 C++的类和对象之后，我们看到了，怎样在一个已存在的类的基础上使用继承来创建一个新类。接着看到面向对象编程的一个重要概念：多态。它可以通过指针，这个指针的类型是某个父类的，就可以来代表不同类型（指不同的子类）的对象。虚函数与多态形影不离，因为它可以让我们通过指定指针的类型或者对象的类型来调用某个

函数。

于是我们检查以确定怎么使用继承，何时使用继承，正确使用继承的方法，看看继承有没有其他的替代方案，查看使用继承的潜在劣势等。特别地，还仔细了解了继承和虚函数是如何实现的等细节，还有它们可能导致的性能损失。

1.11 阅读建议

这里有一些很不错的介绍 C++的书籍。你也许想浏览一下，复习一下本章中出现的一些基本概念，如果过去没有 C++的开发经验，也可以从头开始学习它。

Eckel, Bruce, Thinking in C++. Prentice Hall,2000。也可以在线阅读，网址是：http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html。

Lippman, Stanley B., C++ Primer,Addison-Wesley,1998.Cline,Marshall,C++FAQ Lite, http://www.parashift.com/c++-faq-lite/。

下面的书籍就怎么使用或者不用继承给了很好的建议：

Cargill, Tom, C++ Programming Style Addision-Wesley, 1992.

Murray, Robert B., C++ Strategies and Tactics, Addison-Wesley, 1993.

这里还有一本书，里面深入介绍了继承和虚函数的实现细节，这也是一本很好的参考书。

Lippman, Stanley B.,Inside the C++ Object Mode, Addison-Wesley,1996

第2章 多重继承

本章将讲述：

- 使用多重继承
- 多重继承的问题
- 什么时候使用，什么时候避免多重继承
- 多重继承的实现（高级话题）
- 性能分析（高级话题）

多重继承是C++的另一个新内容。单继承可以让我们从一个父类去创建一个新类，多重继承扩展可以让我们从两个或更多个父类派生出一个新类。它没有单继承广泛的使用，但它也有单继承同样的问题。但是运用得当的话，它将是你设计的得力助手。需要特别指出的是，多重继承的常用技巧，比如抽象接口，在实际的开发当中，会特别有用。

2.1 使用多重继承

考虑一个简单的设计问题：怎样用已有的方法实现它，用多重继承又是怎样解决问题的。比如说，设计一个基本的类：GameObject。这个类是最基本的，所有类型的游戏对象都要从它继承，敌人（enemy）、触发器、镜头等。特别地，有两个方面是必须要实现的。那就是，所有的游戏对象都可以接收消息，所有的游戏对象还都可以挂到一棵树上（游戏中游戏对象一般是有层次的，实际开发中一般用树形结构来实现）。先不考虑GameEntity的其他方面，怎样实现这两个需求呢？

2.1.1 一体化的方法

最明显的办法是在GameEntity类里实现这两个需求，只要加两个函数，一个接收消息，另外一个用来实现把游戏对象挂到树上，并且可以挂到树的任意结点上。

一般而言，一个问题用C++实现往往有很多不同的方法，这个问题也是这样。不幸的是，第一个映入脑海的，或者需要敲键盘最少的方法，往往不是最好的。一体化的方法很简单，也很直观，但是它有些缺陷。

类的简洁性是一把双刃剑。不错，一方面，它很容易添加新的函数而无须创建一个新类，很容易改变继承关系，也很容易修改类的结构；但是，另外一方面，每一次添加新东

西，GameEntity 类就变得越来越大，复杂性越来越高。过不了多久，一个 GameEntity 这样的基础类会膨胀到代码量很大，函数混乱很不利于使用，也不利于维护。虽然短期上来说，很简单地就实现了目标，但是从长远的观点来看，这会导致事情变得很复杂。

这种办法的另外一个问题是代码的到处复制。GameEntity 是惟一需要接收消息的类吗？可能 Player 类也需要接收消息，其自身还不想成为一个游戏对象。那怎样挂到树上？也许，其他的对象，比如场景节点或者骨骼动画，可能组织的很简单。但是很遗憾，与不良的软件工程经历一样，代码不好复用。仅仅是需要时，就把相关的代码复制过去，这实在不能算作好办法。因为，当程序改变时，这将导致很麻烦的局面，因为修改很多处代码，将来代码的维护也是问题。

2.1.2 容器的方法

很明显，每一个这样的新概念都应当用其新类来表示。这个例子中应当有两个类：MessageReceiver 和 TreeNode。问题依然存在：这两个类如何与 GameEntity 关联起来？

游戏对象类应当包含这些游戏对象的一个实例，还要提供使用它们的接口函数。这个办法叫做“容器”（containment，参见第 1 章），因为 GameEntity 对象包含一个 MessageReceiver 对象和一个 TreeNode 对象。就像在接下来的章节中可以看到的那样，通常情况下，这是一个很好的办法。这样的好处是，代码可以高度重用，无须对要扩展的类做较大的、较复杂的改动。

惟一的缺陷是有太多接口函数，这些接口函数的主要目的是调用另外的对象的成员函数，我们必须创建和维护这样的函数。特别是，如果接口改变频繁，这些函数维护起来就比较头疼。而且由于额外的函数调用开支，也会带来性能上的损失。下面就是容器的例子：

```
class GameEntity{
public:
        // MessageReceiver function
        bool ReceiveMessage(const Message & msg);

        // TreeNode functions
        GameEntity * GetParent();
        GameEntity * GetFirstChild();
        // ...

private:
                MessageReceiver  m_MsgReceiver;
                TreeNode         m_TreeNode;
};
```

```
inline bool GameEntity::ReceiveMessage(const Message & msg){
        return m_MsgReceiver.ReceiveMessage(msg);
}
inline GameEntity * GameEntity::GetParent(){
        return m_TreeNode.GetParent();
}
inline GameEntity * GameEntity::GetFirstChild(){
        return m_TreeNode.GetFirstChild();
}
```

像第一个例子那样，使用容器（containment）来实现。不过，不是提供成员函数来与 GameEntity 的内部对象交互，我们直接暴露对象本身，这比维护接口函数要方便的多。然而这样却暴露了太多关于 GameEntity 是怎样实现的细节方面的信息。如果想修改 GameEntity 的实现方式，而新的实现方式又不使用这些类（指 MessageReceiver 和 TreeNode），那么所有使用 GameEntity 的代码不得不修改。这样不好，暂时还是让它们保持私有吧（如图 2.1 所示）。

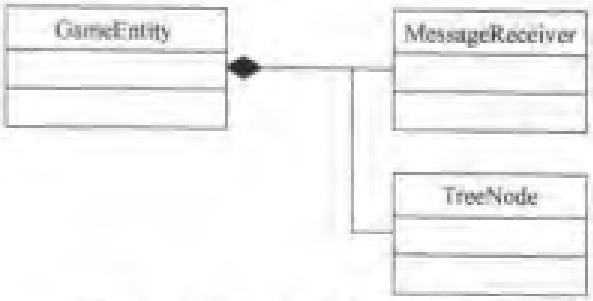


图 2.1　GameEntity 类使用容器实现

2.1.3　单继承的方法

用第 1 章所学的单继承来解决这个问题，大家知道从一个类继承出一个不同的新类很容易，声明 GameEntity 是一个 MessageReceiver，还有它是从 MessageReceiver 继承而来。但是 TreeNode 怎么办呢？从某种程度上说，GameEntity 也是一个 TreeNode。这个概念用单继承不太容易建立模型，我们尝试着创建如图 2.2 所示的继承关系。

表面上看，这个方案可以工作。这个程序也能正常工作，不过它设计的实在是太糟糕了，以后迟早会遇到大麻烦的。TreeNode 是一个 MessageReceiver 吗？不一定非是不可，那为什么非要从 MessageReceiver 继承呢？另外，生成这样的继承树，如果某个类不是从 MessageReceiver 继承的，就没法使用 TreeNode 了。所以说，还是再看看有没有更好的方法吧。

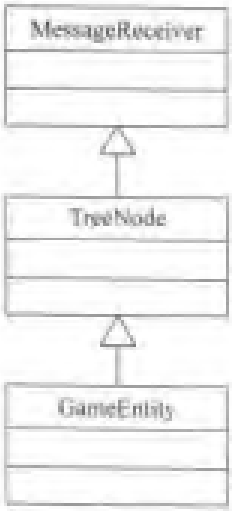


图 2.2　单继承不是永远都好使

2.1.4　多重继承更有利于复用

多重继承是这个问题的解决办法。和单继承的工作原理一样，只是一个类允许从多个父类派生。对本例子来说，可以让 GameEntity 既是从 MessageReceiver 派生的，也是从 TreeNode 派生的，于是这个新类就自动具有父类的接口，具有其成员变量以及成员函数。代码大概是这样的：

```
class GameEntity : public MessageReceiver , public TreeNode
{
public :
    //  Game entity functions…
};
```

对应的继承关系如图 2.3 所示。

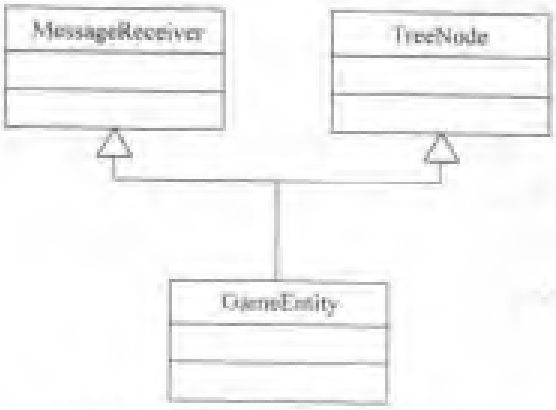


图 2.3　GameEntity 用多重继承来实现

具有多重继承关系的类，声明对象和使用单继承的类声明对象一样：

```
GameEntity entity;
//…
GameEntity *pParent=entity.GetParent();
```

2.2 多重继承的问题

一般来说，引入新特性新功能的同时，会使问题变得复杂，也会带来新的问题。多重继承也不例外。事实上，有些人认为多重继承带来的新问题多于它所解决的问题。下面来看看都有哪些问题吧。

2.2.1 二义性

第一个问题就是它的二义性。如果要继承的两个父类都有一个成员函数，而这两个函数有着同样的名字和参数，那将发生什么？想想上一个例子中，假设 MessageReceiver 和 TreeNode 都有一个公有的成员函数，名字为 IsValid()，该函数程序调试会用到，用来检测对象的状态是否正确。当一个 GameEntity 对象调用 IsValid()时会怎样？结果是一个编译错误，就是由于这个调用具有二义性。

为了解决这个二义性问题，需要在调用函数的时候，前面加上类的名字。如果想调用 GameEntity 内的函数，则需要范围符，像下面的代码那样：

```
void GameEntity ::SomeFunction(){
      if( MessageReceiver::IsValid() && TreeNode::IsValid() ){
          //…
      }
}
```

2.2.2 拓扑图

还有一个更大的问题是拓扑图。考虑一下下面的情形：有一个 LandAI 的类，这个类处理游戏对象在地面上的行走，还有一个类叫 FlyingAI，这个类处理游戏对象在空气中的飞行。游戏设计者提出一个新类型的混血对象，要求新对象既可以在地上走，也可以在空气中飞行。需要创建一个新的 AI 类，一个可能的办法就是让它同时从 LandAI 和 FlyingAI 继承（如图 2.4 所示）。

一切都很好，直到意识到 LandAI 和 FlyingAI 继承于同一个基类。继承关系实际上如

图 2.5 所示。

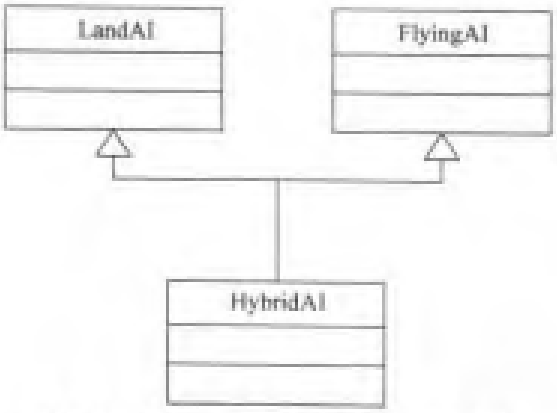


图 2.4 这是一个看起来不好的多继承树

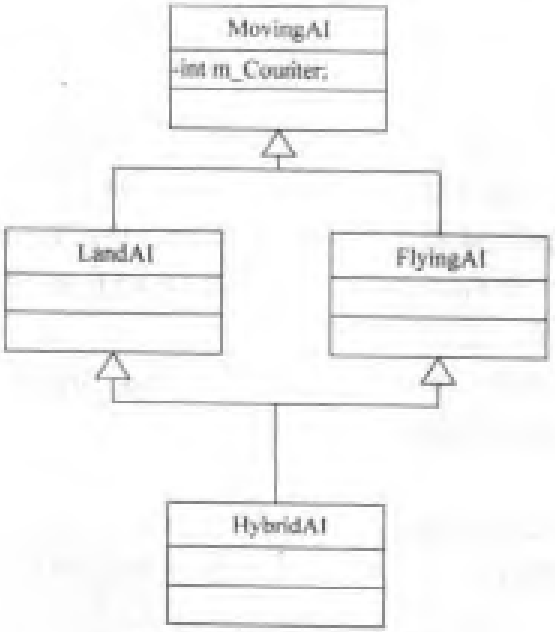


图 2.5 钻石型继承关系

代码里，类的结构大概是这样的：

```
class MovingAI{
      //…
protected:
    int m_Counter;
};

class FlyingAI : public MovingAI{
      //…
};
```

```
class LandAI : public MovingAI{
    //…
};

    Class HybridAI : public FlyingAI, public LandAI{
    //…
};
```

这就是可怕的钻石型继承树，也叫 DOD（Diamond Of Death）。MovingAI 对 HybridAI 来说是父类，但是有两条路径。这种数据组织方式有一些不可预料的后果。

- MovingAI 函数的内容在 HybridAI 出现两次，因为 HybridAI 是由两个不同的类继承而来，而这两个类都有 MovingAI 这个成员函数。同样让人吃惊的还有：HybridAI 也将有两个 m_Counter 成员变量。
- 使用 HybridAI 的 MovingAI 成员函数的一个成员变量时，也存在二义性。需要指定继承的路径，通过这种办法可以访问想要的成员变量。听起来有点多余，因为两个路径都到达同样的变量，但是由于以上的原因，确实不多余。

HybridAI 的一个对象如图 2.6 所示，该图展示了它是如何由父类的不同部分组成的。

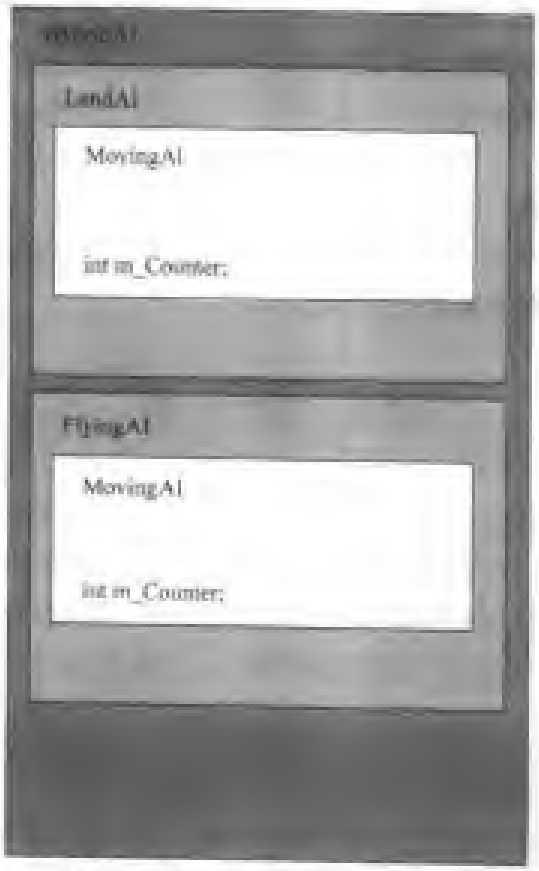


图 2.6 有着多重继承的对象的结构图

为了解决这个问题，C++引入了一个新概念：虚继承，这是一个和虚函数完全不同的概念。虚继承允许父类在其子类对象的结构中仅出现一次，即使在钻石型继承树的情况下也只出现一次。但是虚继承会导致一些运行开销，比如会多用一些内存空间。在这样的情况下，为了像期望的那样能够正常工作，有一些指针（比如虚函数表里的函数的指针）需要调整。此外，虚继承也带来了自己的问题和复杂性。

那最好的方案是怎样的呢？最好是不惜一切代价去避免钻石型继承树的情况。通常，这意味着类设计的不好。而且从长远的观点来看，同它能解决的问题相比较，它会带来更大的麻烦。如果你确信钻石型继承树这种设计对你的程序来说是最好的，那请确信你和你的开发团队都能知道所有这个类的所有细节，以及虚继承的种种副作用。本章末尾部分的阅读建议介绍了一些书籍，可以作为了解虚继承的开始。至于本书的其余部分，将避免出现虚继承和钻石型继承关系的情况。

2.2.3 程序架构

好像刚才说的还不够似的，最后再补充几句。多重继承还能带来一个重要的麻烦：虽然正确的使用了，但是如果一不小心的话，很有可能导致糟糕透顶的程序架构。过分依赖多重继承，从小的程度上说，单继承不会使继承树层次过深，接口不至于过于膨胀，类之间的耦合性也不会太紧。而转向多重继承后，不仅不同的环境下重用某个类太麻烦，而且使得维护现存代码和为代码增添新特性变得困难，也将导致编译和链接的时间一定程度的增加。

多重继承是一个很复杂、也很难运用的工具。只要有可能，最好能找到一个替代方案，比如 composition 模式。仅仅在多重继承在所有的方案中是最好的这种情况下才选用它。本章的后面会详细解释多重继承的最佳方案的例子。

2.3 多　态

和单继承一样的是，可以通过指针（该指针类型为它的某个父类）来引用一个对象。然而和单继承不一样的是，我们需要特别小心，怎样去获取和操纵这些指针。

我们经常把一个指针强制转换成它的父类的指针，像单继承里的那个例子一样。可以使用老式样的 C 风格的强制转换或者更友好的 C++风格的强制转换。

```
GameEntity * pEntity = GetEntity();
MessageReceiver * pRec;
pRec = (MessageReceiver *)(pEntity);        //C cast
pRec2 = static_cast< MessageReceiver * >(pEntity);      //C++ cast
```

然而，当强制转换和多重继承同时出现的时候，事情变得复杂起来。在单继承里面，

惟一要做的就是确保要指向的对象的类型是正确的，直接强制转换就可以了。在多重继承里面，这条路行不通。原因是有着多重继承的对象，它的结构就是 vtable，和单继承的情况有所不同。在单继承里面，vtable 的开头部分对所有这个继承体系下的类都是一样的。派生出的新类可以使用 vtable 的入口。而在多重继承里面，不同的基类在 vtable 里有着不同的入口点，所以，强制转换后实际上返回的是一个不同的指针，而不是它真正要转换的那个。这个问题在本章的剩下部分——多重继承的实现里，再做详细的探讨。

强制转换是怎么实现的呢？典型的实现是通过使用 dynamic_cast，这是一种新的强制转换风格。不像其他种类的强制转换，dynamic_cast 能引入运行期代码，这些真正运行期代码可以完成必要的指针运算，还可以调整在不同的 vtable 中的偏移量。此外，如果强制转换操作了非法的对象或者类的继承体系，dynamic_cast 会返回 NULL。

```
GameEntity * pEntity = GetEntity();
// Normal dynamic cast, Works fine.

MessageReceiver *pRec;
pRec = dynamic_cast<MessageReceiver*>(pEntity);

//Also works fine, but pNode will have a different value
//than pEntity
TreeNode * pNode;
pNode = dynamic_cast<TreeNode*>(pEntity);

//This is not a valid cast because the entity we have is not
//accually a player object. It will fail and return NULL.
Player * pPlayer;
pPlayer = dynamic_cast<Player *>(pEntity);
```

不幸的是，dynamic_cast 不仅会带来一些轻微的性能损失，还要求编译器设置成允许使用 RTTI（Runtime Type Information，运行期类型信息）。也就是说，为了使 dynamic_cast 正常工作，编译器需要创建和保存所有类在运行期的有关信息。虽然对每一个类来说，不会需要太多的内存，但是每一个类都需要这样的信息，所以其占用内存的总和还是很可观的。这太不幸了。因为我们可能不需要为每一个类都保存这样的信息，尤其是那些简单的轻量级的类，比如矩阵类（matrices）或者向量类（vector）。第 12 章将仔细考查编译器默认的 RTTI 系统，并提出改进以适应开发游戏的需要。

2.4 什么时候使用，什么时候避免多重继承

到目前为止，这章对多重继承的介绍，好像使用多重继承没有多少好处，没有什么好

的应用前景。读者也许会疑惑，既然这样，为什么还要花费整整一章的篇幅来介绍它呢？毕竟，目前的结论都是尽可能避免多重继承的使用，好像对它不利。然而，如果应用得当的话，多重继承会是一个很好的工具。

需要牢记的最重要的一点是，不要随意地使用多重继承。因为第一个映入你脑海的方案往往不是最好的。我们已经了解多重继承带来的潜在问题，会导致二义性，增加程序的复杂性，会导致微小的性能损失等。

只要有可能，考虑用容器（containment）作为多重继承的替代方案。大多数的情况下，这个设计会更好。如果容器不可能使用或者使用容器过于繁琐，也不要使用单继承，除非单继承正好合适这种情况。为了解决一个需要多重继承来解决的问题，把一个单继承链弄得混乱不堪，只会把事情弄糟。这样会产生无用的临时类，这些类没有什么真正的意义，而且这些类的存在还会使程序变得难于理解、难于维护。

第 10 章将会详细介绍抽象接口，这是多重继承的伟大应用。只要对类的种类做一些很少的限定，抽象接口就可以使用多重继承，而且没有本章中所说的那些负面问题。抽象接口可以带来以下好处：在运行期切换类的实现，游戏发行后扩展游戏的功能，为程序创建插件，这些都是第 11 章的话题了。

避免在很深的、很复杂的继承体系中使用多重继承。当你想要理清某个功能是在哪儿怎样的，程序的流程是怎样的时候，单继承都会使这很困难了。选择多重继承只会比这更烦。还有，如果继承很深，继承关系很复杂的话，很容易出现可怕的钻石型继承关系（关于这一点前面有介绍）。这样的事情应该避免，甚至不惜一切代价，除非你确实需要那样的设计，除非你非常清楚这样的后果，除非你非常了解虚继承。

关于多重继承的例子，在过去的游戏开发也有，但都比较简单，只是对类的一般性扩展。引用计数器类（reference-counter）是一个很好的例子，该功能可被多重继承自由的使用。从它继承出的成员函数，通过调用 AddRef 和 Release 函数，都会自动地具有引用计数的特性。继承于这样的类很让人放心，因为它简单，它没有基类。

2.5 多重继承的实现（高级话题）

多重继承的实现和单继承的实现很相似，但是复杂性增加了。不幸的是，多重继承不会像单继承那样带来代码的整洁和程序效率的提高。

单继承实现的最优雅的方面之一就是，对象是后向兼容的，可以兼容父类的对象。这是由于以下的原因：单继承只有一个 vtable 指针，派生类新加的其他成员变量都放到后面，正是由于这个原因，父类可以不理会它们的存在。这个特性使得我们只要知道了对象的种类就可以推算出继承关系（如图 2.7 所示）。

在多重继承的情况下，事情变得有点复杂了。由于多态的原因，一个派生出的类的地址只有经过正确的计算后才能确定下来，这和它的父类一样。如果还像单继承那样，仅仅把成员变量追加到后面，只用一个 vtable 是不可能起作用的。因为我们无论如何尝试，都

不可能把它调得像它的所有的父类一样（意思是当只有一个父类的时候，比如单继承的情形，可以把派生类新的成员变量追加到后面，但多重继承由于有不止一个父类，所以一般不可能实现这一点）。

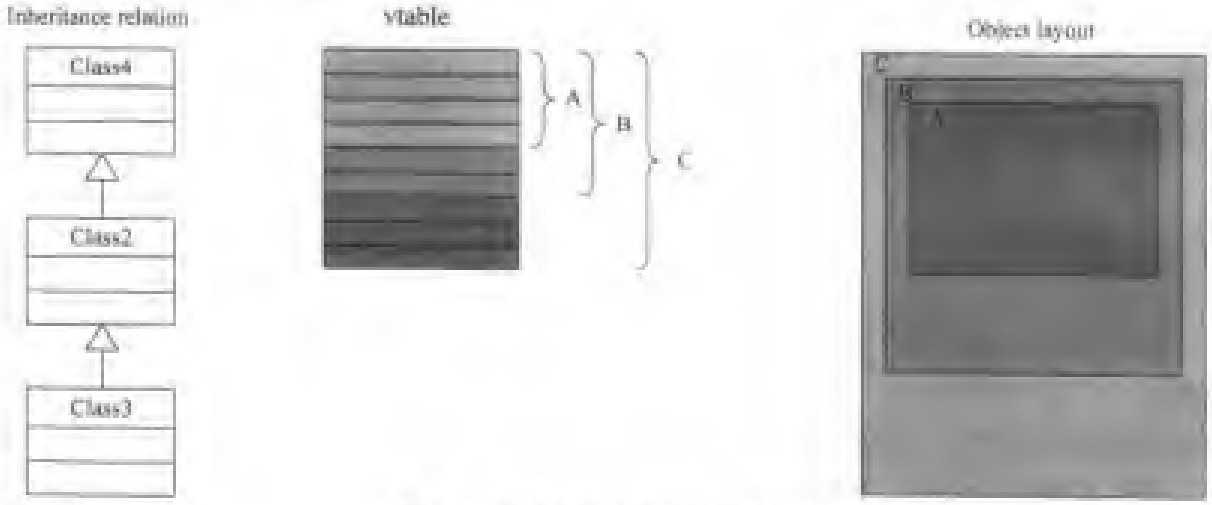


图 2.7　使用单继承的父类和派生类

连同数据一起，我们必须为每一个父类追加一个 vtable。现在把一个派生的对象强制转换到它的父类可以实现了，只要为 vtable 指针指明一个偏移量就可以了（如图 2.8 所示）。

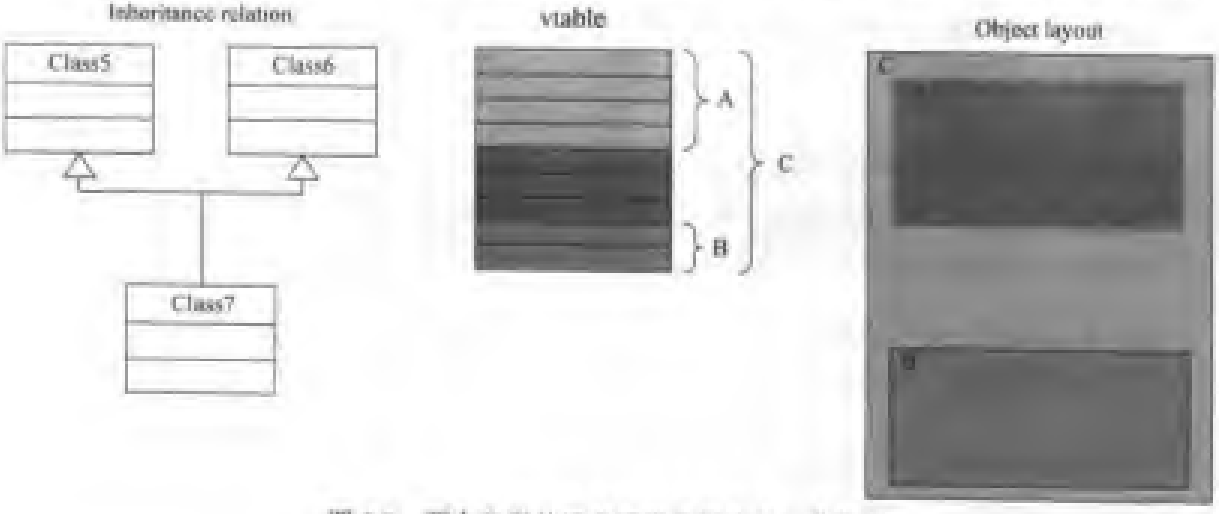


图 2.8　两个父类使用多重继承派生出一个新类

从内存的方面来看，多重继承会为每一个父类的每一个对象额外添加一个指针。这对于现今一般 PC 内存都比较大的情况来说不是个问题。但是需要牢记这一点，也许有一些对象会成千上万次的被创建，要记得这些额外的内存消耗。对于单继承来说，只有在要继承的父类存在虚函数的情况下，vtable 指针才是必要的。其余的情况，只要把成员变量追加到后面就可以了。

最后需要指明的是，派生类的顺序（指为派生新类而追加的信息）是和具体实现相关的。大多数编译器会把它们按照继承的声明时的顺序而追加，但也有一些编译器会打乱这

个顺序，这是为了获取一些性能上的改进。

2.6 性能分析（高级话题）

多重继承和单继承一样有着性能的特点，不同之处有二，这主要是指强制转换以及第二父类的虚函数。

2.6.1 强制转换

在单继承的例子中，指针的强制转换是很容易的。转换仅仅是识别出一个对象属于编译器处理的哪个具体的类。编译器可以保证那个指针的所有操作是合法的，编译器可以保证任何虚函数都使用了正确的 vtable。

对于多重继承来说，事情就变得复杂了。派生类的对象是由多个相关的类的对象组成的，每一个对象都有自己的 vtable。当在不同的种类之间作转换时，需要对指针做一些调整。特别是，当从一个派生类转换到第二个父类（或更靠后的父类），或者相反的操作，需要计算其指针的位置，为了指向对象的准确位置需要为指针附加一个小的偏移量。

对象的指针转换成第一个父类的类型的时候，对指针没有任何影响，这个操作是允许的，这和单继承的情况一样。从派生类转换成第一个父类的操作，对指针没有任何不良影响，如图 2.9（a）所示。

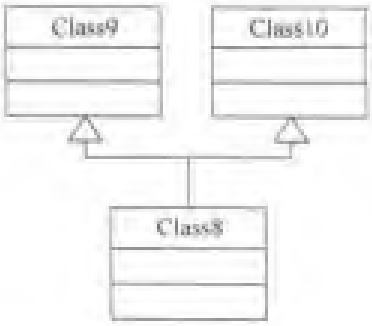
```
Parent1 * pParent1 = new Child;
Child * pChild = dynamic_cast<Child*>(pParent1);
assert ( pParent1 == pChild);              //Unchanged
```

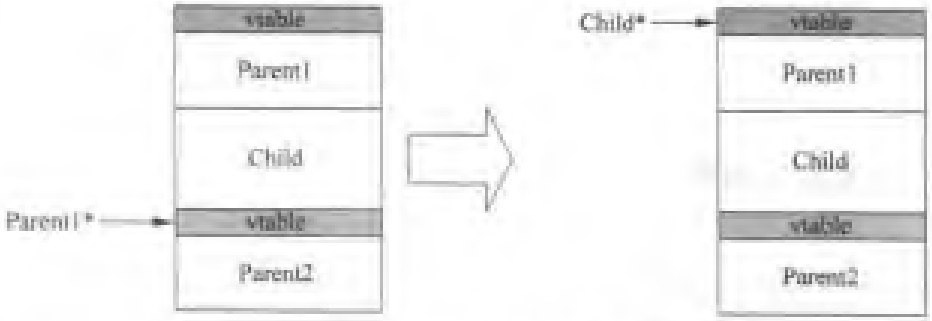
如果不是从第一个父类转换到派生类的话，那就会改变指针。这种情况下，会为指针加上一个正的偏移量，从而使它指向对象的真正开始部分，如图 2.9（b）所示。

```
Parent2 * pParent2 = new Child;
Child * pChild = dynamic_cast<Child*>(pParent2);
assert ( pParent2 != pChild);          //Not the same!
```

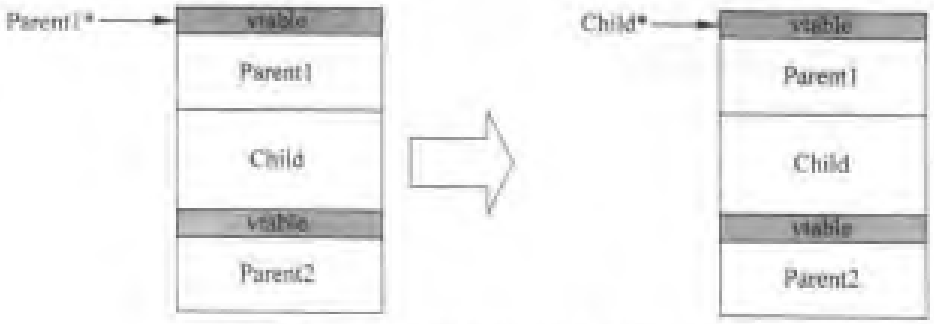
最后，从派生类转换到第二个父类也会改变指针，会为指针加上一个小的偏移量，从而使得它指向父类的对象的对应部分，如图 2.9（c）所示。

```
Child * pChild = new Child;
Parent2 * pParent2 = dynamic_cast< Parent2*>(pChild);
assert ( pChild != pParent2);          //Not the same!
```

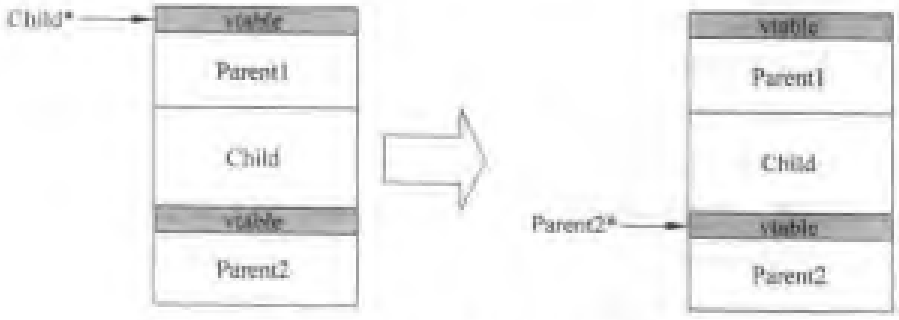




（a）从第一个父类转换为子类



（b）把第二个父类的指针转换为子类的类型



（c）从子类的指针类型转换为第二个父类的类型

图 2.9　强制转换

那么这样的额外指针偏移量的计算会带来性能上的多大损失呢？不会很大。为指针进行的加减偏移量的计算是在编译期完成的，所以说，甚至不会额外地存取内存中的数据（内存中的数据访问可能会带来数据缓存更新过于频繁等问题）。访问 vtable 的开销，还有最后的函数调用的开销才是大头，为指针加个偏移量的操作的开销根本算不上什么。

更可怕的性能损失来自于 dynamic_cast< >这个调用本身。在运行期，程序需要确定把一个初始类型的指针转化成另外一个类型是否可行。为了实现这一点，需要考虑要强制转换的指针的类型，引用了这个指针的对象的类型，还有最终转换后的那个类的类型。取决于具体的实现，继承体系越庞大，越复杂，这个操作也就越费时。第 12 章会仔细研究动态转换和运行期类型信息（RTTI）的详细情况，还要实现一个自定义的 RTTI，以获得较好的性能特性。

2.6.2 第二父类的虚函数

与多重继承的性能有关的开销还有一种，而且与 dynamic_cast 的调用无关。如果调用了一个虚函数，而这个虚函数不是第一个父类的虚函数的话（属于第二个或者其他的父类），那就会涉及另外一种比较小的性能开销。

再看一下图 2.8，注意派生的对象是怎样具有多个 vtable 的。无论什么时候，一个 vtable 的函数被调用，只要它不是 vtable 中的第一个函数，在真正被调用之前，指针都需要调整。在实现这之前，指针的调整和转换操作的情况一样，也就是本章前一部分所描述的那样。很幸运的是，这次没有涉及到 dynamic_cast 调用，所以大多数情况下，性能开销很小，甚至可以忽略。

2.7 结　论

这一章考查了多重继承的概念。多重继承使得我们可以在两个或更多个父类的基础上，创建出一个新类。这一点和单继承相似，只是有着多于一个的父类。而且多重继承的运行确实和单继承非常相似。

不好的方面是，多重继承有一些问题：由于钻石型继承树带来的二义性，需要虚继承来解决这个问题，而这会带来微小的性能损失。最糟糕的是，代码的编译原本很简单，现在却由于使用了多重继承变得相当复杂。

由于以上的问题，当设计程序的时候，最好看看有没有其他的办法能代替多重继承。然而，有一个比较好的多重继承的应用实例，抽象接口以及抽象接口的应用，比如插件。在后面的章节里将会详细说明抽象接口及其应用。

2.8 阅读建议

下面的书籍都谈到了多重继承的一般性建议，怎样使用多重继承，如何避免滥用多重继承等：

Cargill,Tom, C++ Programming Style, Addison-Wesley, 1992.

Meyers,Scott, Effective C++ Second Edition, Addison- Wesley, 1997.

Meyers,Scott, More Effective C++, Addison- Wesley, 1995.

Murray,Robert B., C++ Strategies and Tactics, Addison- Wesley, 1993.

关于多重继承的具体实现的所有细节，包括虚继承的探讨，可以参考下面的书籍：

Lippman, Stanley B., Inside the C++ Object Mode, Addison-Wesley, 1996.

第 3 章　常量及引用

本章将讲述：

- 常量（const）
- 引用
- 强制转换

本章讲述一些 C++的新概念，这些概念在 C 中是没有的。它们不是最基础的概念，也不是最重要的概念，如继承或者模板，模板将在第 4 章中讲述。相反，它们是受限很厉害的概念，而且它们也不会对程序结构产生大的影响。本章更像一个插曲，位于复杂的多重继承之后、让人头疼的模板之前。

特别地，本章的话题有：常量，它可以使我们把一个对象标记为只读；引用，一种更安全、也更友好的指针形式；还有新型的强制转换，尽管有点冗长、累赘，但它可以使得程序清晰，比 C 风格的强制要方便的多。

所有这些概念在日常的编程中都很有用，本书的很多地方都用它，当然也出现在现今大多数的 C++代码中。熟悉这些新概念以及如何使用它们，才是本章的重点。

3.1　常　　量

const 关键字不是 C++的新东西，但是你应该比通常的 C 程序里的 const 多花点时间。除了通常 C 语言中的意义外，它还被扩展了，它可以修饰引用和成员函数。

3.1.1　概念

const 的概念非常明确，表明被标记为 const 的部分，在程序运行过程当中是不能改变的（即常量）。“部分”这个词有点意义不明，这么用是为了使定义更为通用，可以适应各种各样的可以标记为 const 的情形，后面还会看到。

从 const 自身来说，它很有用。使得 const 是一个较为突出的特性的原因是，编译器会采用像标记一个类是私有还有保护类型一样的方式，该方式可以确保：如果有人想要改变一个标记了 const 的内容时，编译器就立即报告一个编译期错误。编译期是发现这类错误的最佳时段。发现的越早，改正的就会越早。先看一个 const 类型变量的例子。

```
const  int MAX_PLAYERS = 4;
const char * AppName = "MyApp";
```

以上代码中的变量被标记为 const，于是所有的对这个变量的修改将导致编译期错误。

```
MAX_PLAYERS = 2;        // Error, MAX_PLAYERS is declared const
```

有经验的 C 程序员不会感到怀疑，使用#define 编译预处理指令也有类似的功能。从功能上来讲，它们是相似的。但是使用一个 const 类型的变量，编译器会对变量做 C++类型安全检查，这将有助于查错，有助于发现潜在的问题。而使用#define 仅仅是代码的替换，缺少类型检查。通常的原则是，应当尽可能地依靠编译器来查错。如果编译器可以胜任这种繁重的工作，还是把时间花在其他地方，而不是花在跟踪奇怪的类型转换问题上。

使用 const 类型的变量的另外一个优点是，编译的时候，它们被加入到符号表，这意味着，在调试器里也可以使用它们。由于这个原因，在调试器里可以查看 const 变量的符号名，调试变得相当容易。而如果使用#define 的话，究竟它代表着什么，就只有瞎猜了。那些需要在 Win32 下调试代码，查看错误和需要弄清某个标志具体代表什么的人应该感谢这个优点了。

3.1.2 指针和常量

指针和 const 联用的时候，其意义可能有点让人琢磨不定，看以下 4 种可能性：

```
int * pData1;
const int * pData2;
int * const pData3;
const int * const pData4;
```

第一个和最后一个是比较简单的。很明显，pData1 是一个非 const 指针，它指向一个非 const 数据，意思是说，可以随意改变这个指针和它指向的数据。最后一个，也就是 pData4，正好相反，既不能改变其指针，也不能改变它指向的数据。但是另外两个是什么意思呢？

const 限定的是紧跟其右的内容。所以，pData2 是一个非 const 指针，它指向一个 const 整数。另一方面，pData3 的左边有一个 const 限定，所以这个指针是 const 的，但是它指向的数据不是。在头脑中适当地添一些括号可以使事情更清晰一些。

```
//不是C++代码，只是为了帮助理解
(const int *)  pData2;
int * (const pData3);
```

好像这还不够复杂似的，C++新增了一个语法变种来表示相同的概念。若不掌握这个，将来面试的时候对付那些复杂的问题可就困难了。想想下面的第五种形式代表什么意思？

```
int const * pData5;
```

这个语句，句法上是正确的。但是 const 位于数据类型和星号之间。像刚才那样，在脑子里添上括号，(int const *) pData5，这就可以得到正确的答案了。这是一个非 const 的指针，它指向一个 const 的数据。这个例子和 pData2 没有区别，仅仅是 const 的位置有细微的差别。这个风格的写法不常用，但是有时候会碰到。所以至少知道存在着这样一种替代形式。

3.1.3 函数和常量

const 最有用的用途之一就是用来标记函数的参数和返回值。有时候需要传递函数的一个参数，如果这个参数太大或者复制的代价太昂贵（指内存），常常传递一个指针（或者引用，下一章将会讲到）而不是它本身。使用指针能避免复制的开销，非常有效。然而，这样做改变了程序的行为。本来是要传递原始数据的一个复制，但是现在，为了运行速度快一点，传递了原始数据本身。从参数传值改变成传引用，这主要是出于性能的考虑。

由于性能的原因，给函数参数传递的是数据的指针，如果不用 const，我们没有办法知道，参数传递进去后，函数是否改变了原始数据。这个特性非常重要，尤其是需要维护的代码较多的时候。更糟糕的是，函数最开始可能仅仅想读取传入的数据，但是函数下面的代码后来可能会被改成需要改变这个数据。如果程序的其他地方假定这个数据不能改变，这可能会成为一个灾难性的事件。

使用 const 就可以解决这样的问题。const 可以保证不能改变数据变化，使代码更清楚。编译器会检查标记了 const 的数据是否被改变，如果改变了，就报告编译错误。const 还可以使得在阅读源代码时很便捷地知道指针（或者引用）参数的意图。

```
//Clearly pos is read-only
void GameEntity::SetPosition (const Point3d * pos );

//Vector entities will not be modified
int AI::SelectTarget (const vector<GameEntity*> * ents);
```

如果一个参数传给一个函数的时候是传值的，那就没有必要使用 const。我们传给函数的是数据的备份，所以不存在避免函数改变这个参数的问题。这是一个实现的细节问题，调用这个函数的代码无须关注这个问题。

同样的概念也适用于返回值。想象一下，一个函数需要返回的是备份起来开销很大的数据，比如字符串或者矩阵 matrix。一般的优化是不要返回对象的一个新备份，而是直接返回对象的指针。现在可以通过函数返回的指针来改变对象的内容。如果开始的意图就是这样，那一切 OK。然而，也许在设计中，我们并不想有人能够这么直接地改变数据。这种情况下，设计就有了一个潜在的“天窗”（指设计缺陷）。考虑一下下面的 Player 类的例子：

```
// Player class with some constness problems
class Player
{
public:
      void SetName(char * name);
      char * GetName();
      //…
private:
     char m_name[128];
};

void Player::SetName (char * name)
{
     ::strcpy(m_name, name);
}

char * Player::GetName()
{
     return m_name;
}
```

乍一看，好像一切正常。可以设置名字（也许是响应玩家在用户界面上输入他的名字），也可以取出这个名字，打印在积分榜上或者通过网络发送聊天消息给其他玩家。这个类可以像预想的那样正常工作。

然而，就像代码中写的，这有着很多潜在的问题。所有的人都可以调用函数 GetName，得到玩家对象的真实的字符串指针，不仅可以读取，也可以没有限制的改变它。这是问题吗？很有可能。程序的设计是让人通过调用 SetName 函数来改变玩家名字。也许新名字要设置为网络上所有的其他玩家，或者也许新名字仅仅需要在用户界面上改变，无论哪种情况，通过改变调用 GetName 函数返回的指针的内容，都不能得到意想的结果。这也是一个很难跟踪的 BUG，因为它不会导致程序崩溃或者很快有不正常的现象发生。事情变得很诡秘，这样的 BUG 特别难以跟踪。

去除这个潜在问题的一个办法，就是每一帧都检查玩家的名字是否被改变，如果改变了，就做这个改变的对应的响应。但是这需要大量的工作，也很复杂。

可以在写文档的时候仔细说明这个函数，解释清楚 GetName 函数返回的字符数组的指针，不希望直接被改变。但如果是工期太紧，程序员不得不匆忙交付代码的时候，还是常常有人误用这个函数。

另外一个好一点的办法是，让编译器来保证这样的对象不能改变。可以使用 const 作为函数返回值类型的修饰语。

```
// A better Player class without const problems
class Player
{
public:
   void SetName ( const char * name);
   const char * GetName();
   //…
private:
   char m_name[128];
};

void Player::SetName (constchar * name)
{
    ::strcpy(m_name, name);
}

const char * Player::GetName()
{
    return m_name;
}
```

现在，改变玩家名字的惟一途径就是调用 SetName 函数，而且代码会得到极大的简化。如果有人忘记这一点，还指望通过改变 GetName 返回的指针来改变名字，编译器会立即通知他这个错误。

不幸的是，Player 类仍然遗留一些问题，就在 GetName 和 SetName 这两个函数里。一个更简洁的办法是使用引用而不是指针（下一章里会讲述），应该使用 string 类而不是字符指针（见第 9 章，STL 里的 String 一节）。但是即使这么改了，const 的使用还是不变。

3.1.4 类和常量

到目前为止，本章回顾了 C 语言的关键字 const 是怎样使用的，但是本书是讲述 C++的，那为什么叫回顾呢？这是因为：第一，因为 const 在 C 语言中应用的并不广泛，或者说没有 C++中应用的那么频繁；第二，C++又扩展了 const 的应用领域，可以用来限定类的成员函数，所以在了解新的应用领域之前，最好还是熟悉一下过去的用法。C++允许标记一个类的成员函数为 const，例如：

```
// An even better Player class
class Player
{
public:
```

```
    void SetName ( const char* name);
    const char * GetName() const;
    // ...
private:
    char m_name[128];
};

const char * Player::GetName() const
{
    return m_name;
}
```

注意 GetName 这个成员函数的说明和实现都被标记为 const。成员函数标记为 const 意味着调用该函数的时候，不能改变对象的状态。

本例中，通过标记 GetName 为 const，我们告诉程序的读者（还有编译器），调用 GetName 这个函数不会改变 Player 这个类中所有的成员变量。确实是这样的，注意函数仅仅是返回一个指针。另一方面，SetName 这个函数没有被标记为 const，这是因为 SetName 改变了 Player 对象的内部的状态，也就是改变了名字。

由于同样的原因，标记变量或者函数的参数为 const 也很重要，实际上，是更重要。我们为函数附加了更多的信息来说明函数的意图，这样的话，这些代码对其他程序员更具有可读性，而且会通知编译器强制执行这个准则（被 const 限定的东西不能被改变）。

编译器实际上也很聪明。如果试图实现 GetName 这个函数返回一个非 const 的字符指针，编译器会检测到，并报告一个编译错误，因为编译器发现，被标记为 const 的成员函数竟然允许调用该函数的人改变内部的成员变量。所以返回值的 const 限定和成员函数的 const 限定如影随形，一起出现。

这同样也适用于调用其他函数，一个成员函数被标记为 const，它就不能调用一个非 const 的成员函数。换句话说，一个 const 的函数不能改变对象（它所限定的对象或者调用任何其他对象的非 const 的函数）里的数据。

这个结论是非常重要的。这意味着，要想很有效的使用 const，就必须在尽可能多的地方使用它。如果只有一些类的成员函数被标记为 const，这些成员函数将不能调用程序的其他没有标记为 const 的部分。在使用老一点的库的时候，如果这些库没有正确地使用 const 的话，可能会有些烦人。幸运的是，可以抛开 const（本章的以下部分会讲述这个问题）。不用说，仅仅是在非常必要的情况下才抛开它，要不然，就相当于放弃了编译器提供给我们的做有效性检查的种种好处。

3.1.5 常量函数也可以改变成员变量

注意在上面讨论的过程中，谈到了一个对象的“状态”（status）一词。但是并没有确

切地说明“状态”究竟是什么意思？编译器会了解字面上的意思，它把“状态”理解为要改变的对象的成员变量。通常，这个解释和我们想要的正好符合，所以它也能正常工作。

然而有时候，有的成员变量并不能反映该对象的状态，或者不是逻辑上的状态。这种情形常常遇到，尤其是当一个对象保存有一些关于具体实现的内部信息。一个简单的例子就是一个对象有一个变量，该变量记录了某个特定的查询被调用了多少次。

在 Player 的那个例子里，也许每一个 Player 的对象都想保存 GetName 函数被调用的次数。第一个想法就是，每一次该函数被调用的时候，递增一个内部的计数器变量。但是 GetName 函数被标记为 const 了，所以试图改变任何成员变量都会导致编译错误。可以不把这个函数标记为 const，但是这没有一点意义。对所有的目的和意图来说，在函数被调用之前，Player 对象有着同样的状态。为什么使用 Player 类的人会关注我们是否保存有对象内部的统计数据呢？

还有更复杂的例子呢，一个对象隐藏了一些数据。也许对象有太多的数据，所以它只想在需要的时候才加载和卸载。对外部世界来说，对象应当显现它的所有数据。所以这个简单的查询函数应该被标记为 const，即使在内部将加载很多数据。

幸运的是，对这个问题，C++里有一个简洁的办法——关键字 mutable。一个被标记为 mutable 的成员变量可以被任何成员函数修改，不管该成员函数是否被标记为 const。因此，可以标记任何不代表对象逻辑状态的成员变量为 mutable，这样也解决了在一个 const 的成员函数里改变成员变量的问题。

回过头来看看 Player 类的例子，增加一个 mutable 的变量记录 GetName 函数被调用的次数。

```
class Player
{
public:
   void SetName ( const char* name);
   const char * GetName() const;
   // …
private:
   char m_name[128];
   mutable int m_TimesGetNameCalled;
};

const char * Player::GetName() const
{
    ++ m_TimesGetNameCalled; // OK because it is mutable
    return m_name;
}
```

3.1.6 关于常量的建议

关于 const 的最好的建议，就是尽可能多地使用它——任何地方：变量，参数，返回值，还有成员函数。用得越广泛，也就越有用，使用起来也就更容易。

使用 const 没有一点副作用。它可以使代码的意图更清晰，其他程序员读起来更方便，还会让编译器做额外的检查。惟一需要做点零活的情形是，我们需要修改现存的代码库里的代码，开始使用 const，一直到代码里正确地标记为 const 的部分达到一定程度。一旦大多数代码被正确地转化成使用 const，这是很划算的。

3.2 引 用

引用其实很简单，它是对一个对象名字的替代物。所有对引用的操作将直接影响到被引用的原始对象。很有意思的是，如此简单的概念竟然可以成为管理复杂问题的一个相当有用的工具。

引用的语法很简单，除了使用&符号来表明它们是引用外，它们的行为和通常的对象没有什么差别。引用使用起来和内置的数据类型以及对象都是一样的。

```
int a = 100;
int & b = a;                    // b is a reference to a
b = 200;                        // both a and b are 200 now

Matrix4x4 rot = camera.GetRotation();
Matrix4x4 & rot2 = rot;         // rot2 is a reference to rot
rot2.Inverse();                 // inverses both rot and rot2
```

引用和指针很相像。它们都代表一个对象，所有针对它们的操作将直接影响它们代表的数据。而且，创建一个对象的引用和创建一个指针一样，都是效率很高的操作。

3.2.1 引用和指针的对比

然而，在引用和指针之间，还是有些重要的，而且是非常重要的差异的。

- 引用使用起来在语法上和使用对象一样。不使用->操作符来取得指针，不使用->来访问成员函数和成员变量，引用使用点号（.），和通常的对象很相似。
- 引用只能被初始化一次。指针初始化时可以指向一个特定的对象，在需要的时候，可以改变为指向其他不同类型的对象。引用则不行，一旦它们被初始化为一个特

定的对象，它们就不能再改变了。从这个意义上说，引用像 const 的指针。

- 引用必须在声明的时候初始化。和指针不同的是，我们不能创建一个引用，过一会再初始化它，必须立即初始化。
- 引用不能为空（NULL）。这一点是由于以上两点决定的。因为引用必须立即初始化为一个实在的对象，而且以后不能再改变。它们不能像指针那样可以为空（NULL）。不幸的是，这并不意味着它们引用的对象总是有效的。删除一个被引用的对象是有可能的，或者“欺骗”引用，把它所引用的指针指向空（NULL），这两种情况下，引用将为空（NULL）。
- 引用不能像指针那样 new 或者 delete。这个意义上来说，它们像一个对象。

3.2.2 引用和函数

引用的两个主要用途是参数传递和为函数返回值。上一章讲述的 const 指针可以作为函数的参数。我们已经知道，出于效率的考虑，当对象较大时，传递它的指针而不要传递整个对象的备份。虽然使用 const 的指针可以解决所有的潜在问题，但不得不为一个指针而改变一个对象，这显得有点笨拙。引用可以解决这个问题。通过向函数传递一个 const 的引用的参数，可以实现传值同样的目标，这并不会导致性能上的任何损失，而且和传值使用同样的语法。注意所有的关于使用 const 指针的建议，同样适用于引用。

```
Matrix4x4 rot;                  // Relatively expensive object to copy
//...
entity.SetRotation(rot);        // Cheap call, No copying

void GameEntity::SetRotation ( const Matrix4x4 & rot)
{
   if (!rot.IsIdentity)
         //-
}
```

引用也可以应用在函数需要返回一个对象的场合，这种方式很有效。对返回的引用的所作所为，我们需要小心谨慎。这是因为，如果指派返回值为一个对象，就需要这个对象的一个备份。如果仅仅是想要保存那个引用一会儿，来做一些运算，必须把它保存到一个自身的引用中去。

```
const Matrix4x4 & GameEntity::GetRotation() const
{
   return m_rotation; // Cheap, It's just a reference
}
```

```
// Watch out. This is making a new copy of the matrix
Matrix4x4 rot = entity.GetRotation();

//This just holds the reference. Very cheap.
const Matrix4x4 & rot = entity.GetRotation();

//We can pass the reference straight from a return
//Value into a parameter too. Very efficient also.
camera.SetRotation(entity.GetRotation());
```

然而，当使用指针的时候，我们需要保证，要返回的引用所引用的那个对象，当函数执行完的时候，没有超出范围。最常见的情况就是，返回一个引用，它引用了的那个对象是在栈（stack）里创建的。这么做，返回值引用的将是一个无效的内存地址，一旦使用了这个引用，程序就会崩溃。可喜的是，大多数的编译器可以探测到这种情况，并且报告一个警告。

即使没有了解关于引用的这些情况，要想完全地不用它，几乎是不可能的。你将看到引用出现在复制构造函数和二进制操作运算中。所以至少要熟悉它们的用法。很乐观的是，本节的剩下部分会说服你，在自己的代码里使用引用。

```
//Copy constructor
Matrix4x4:: Matrix4x4 (const Matrix4x4 & matrix);

// Binary operator
const & Matrix4x4 Matrix4x4::operator *(const Matrix4x4 & matrix);
```

3.2.3 引用的优势

如果引用像指针那样，只是在使用上有一些语法的不同，那为什么还要使用它呢？为什么不坚持使用指针呢？这有些原因，即使到目前为止，编译器关注的多是指针。真正的原因在于程序员使用它，可以使编码变得更自在一些。

使用引用的第一个优势在于它们的语法。即使最有经验的C程序员也得认可，使用引用比指针代码更简洁一些，至少不要到处使用->了。下面的代码哪个更有可读性？

```
// Using pointers
position = *(pEntity->GetPosition());

// Using references
position = entity.GetPosition();
```

引用的第二个优势在于，它们不像指针那样可以为空（NULL）。通常，它们必须指向一个对象，至少是编译器认为它是一个对象。在任何一种情况下，传递一个非法的引用远比传递一个空指针困难得多。

从某种程度上说，引用指向一个合法的对象，并不意味着对象总是有效的。同样的问题，指针也存在，这就是所谓的“悬指针”问题。这种情况下，我们保存了一个指针，这个指针却指向了一个被释放的内存地址，指针本身没有改变，有BUG的代码可能会递过该指针取数据。然而，回忆一下，引用必须在第一次被创建的时候初始化，而且不能改变其引用的对象。于是，保存一个引用常常很难，因为它指向内存中的对象，当该对象被销毁后，引用也就不存在了。大多数的情况是引用使用在栈里，当出了这个范围后，引用也就不存在了。所以该问题得到一定程度的缓解。

引用的另外一个优势是，至于引用指向的对象是否被使用引用的代码所释放，这是毫无疑问的。这只能通过指针来进行，引用做不了。因此，如果使用引用，就要保证其他人释放它，这样才安全。

引用所有的这些优势可以总结为，使用引用是操作对象的一个稍微高层的方式。它允许我们忘记内存管理的细节，对象的归属，所以可以把精力集中放在要解决的问题的逻辑上。毕竟，我们需要时刻知道，我们为什么写程序是第一位的问题。处理底层细节的时候不要展示技术的强大，也不要使用C++的最新特性。我们是在写一个伟大的游戏！我们需要更多的关注游戏，更少的关注内存泄漏等技术细节，这样我们的游戏才能完成的快一些，游戏才会更稳定一些。

有些人会争辩说，使用引用而不是指针的一个副作用是，究竟对象是传引用还是传值，从调用的代码上看，不太容易看得清楚。可以这样驳斥，调用代码知递对象究竟是传值还是传引用，其实实际上没有多大用。要点在于它们是否可以被函数所改变，这取决于该引用（或者该指针）是否被const标记了。在这两种情况下，只要看一下函数的声明就可以了。现在的开发工具一般都集成有代码浏览工具，查看这样的信息很方便，只不过是点一下鼠标而已。

另外一个细微的但是很明智的建议，也经常被提及，那就是当不允许函数改变某对象时，用const限定的引用就可以了，当需要改变的时候就用指针。至于参数，再说一遍，没有必要通过查看函数的声明来了解函数的意图。尽管某种程度上来说是对的，仍然有这样的机会，即使对象不允许被改变，还是向一个函数传递一个指针。此外，不是每一个人都会遵守约定，这意味着无论何种情况，都需要检查函数的声明。

3.2.4 使用引用的时机

指针是不是过时了，是不是总是使用引用呢？不是的。

一个好的原则是尽量使用引用，引用比较清晰，而且出现错误的可能性很小。然而，有些场合，使用指针仍然很有必要。

如果一个对象需要动态创建或者销毁，应当使用指针。通常，那个需要动态创建的对象的所有者，需要一个指针来保存它。如果有其他人需要使用这个对象，他们可以使用引用，这样就很简洁，也没有义务去释放那个对象。如果对象的所有权需要改变，那就用指针而不要用引用。

有时候需要改变指向的对象。在这种情况下，除非改变了程序的结构，指针是惟一的选择，因为一个引用不能指向另外一个不同的对象。

其他时间则依赖于这样一个事实，那就是指针可以为空（NULL）。或者函数通过返回空指针来表明函数调用失败，或者作为函数的一个可选的参数。引用不能指向一个空对象，所以它不能像指针那样用。一个好的程序设计者依赖于一个有时为空的指针，这本身就是让人质疑的。一个好的方案也许就是，指明函数调用失败，使用引用就行了。

使用指针而不是引用的最后一个原因是：指针的运算。通过指针的运算可以遍历一段内存区域，在使用的指针的类型的基础上，可以解析它的内容。这是相当底层的工作，有可能导致 BUG，风险很大，而且维护这样的代码更像是一场恶梦。只要有可能，要不惜一切代价，避免这种情况。但是如果不得不多次使用指针运算，那没有办法。可能带来的问题是，在循环内部，当以类型安全的方式遍历变得不可接受的时候，代码可能会在这里浪费了太多的时间。然后，才不得不调整为使用指针的算术运算。这样的后果要设法避免。

有研究表明，C++中的 BUG 绝大部分来自于内存泄漏。需要斟酌一个对象究竟是使用指针还是引用，这样的考虑越详尽，程序就会越可靠、越稳定。

3.3 强制转换

转换是指这样的过程，它把数据变成另外一种不同类型的数据。在这种情形，含糊的术语“数据”指的是，从内置的数据类型转换到用户自己创建的对象。编译器有一整套规则来确定转换是否可能，能否成功转换。比如说，指定 float 类型的数据转换到 int 类型就会触发一个转换。

```
int a = 200;
float b = a;     // Conversion from int to float

char txt[] = "Hello";
float c = txt;   // No conversion possible
```

强制转换是加到源程序的一种指令，该指令迫使编译器应用一个特定的转换。在要转换的变量前加上圆括号，在圆括号里写上要转换的类型就可以实现强制转换。

```
int n = 150;
```

```
// n is an integer , which divided by an integer results
// in another integer.  f1 = = 1.0
float f1 = n  / 100;

// n is cast to a float, and when divided by an integer,
// the result is a float , f2 = = 1.5
float f2 = ( float) n / 100; //cast to a float
```

3.3.1 强制转换的必要性

强制转换，是一个经常被大多数程序员所轻视的环节。实际情况是，还有其他更好的、更简洁的办法可以实现同样的目的。每当强制转换一个对象到另外的类型时，相当于告诉编译器，“忘记你所知道的这个对象的类型吧，把它当做一个不同的类型吧。”由于人犯错误的概率远大于计算机，所以最好还是尽可能少地使用强制转换。

然而有时候，强制转换是不得已的事情。使用强制转换的一个理由是，需要和代码的其他部分交互（比如调用其他函数等）。调用接口期望一种特定的对象类型，但是想提供的是另外一种类型，而且该类型已定义好，也是正确的。类型的强制转换可以很好地实现这个目的。这在C语言的库函数里很常见，因为C语言没有继承，也没有多态。

设想一下，一个一般的函数，它使用很多数据类型和一个标志变量来表明怎样解析那些数据。顺便说一句，这是一个很恐怖的函数，这个函数其实可以用C++的一些特性，以一种更为安全的方式实现。但是，这恰好是一个为什么使用强制转化的很好的例子。

```
void  SerializeWrite (DataType type, void * pData);

char txt[] = "This is a string";
:: SerializeWrite (SerializeString, (void *)txt);

float fPitch;
:: SerializeWrite (SerializeFloat, (void *)&fPitch);

const Matrix4x4 & rot = camera.GetRotation();
:: SerializeWrite (SerializeMatrix4x4, (void *)&rot);
```

使用强制转换的另外一个原因是使用了多态。设想一下，扩展了一棵继承树，大多数代码涉及的是通过指针的类型来操作对象。有时候找出正在处理的对象的类型是很必要的，通过它再调用那个对象里的某个成员函数。这通常是不好的设计的一个标志，但经常出现。假设不允许RTTI信息，程序就不得不通过调用一些函数，强制转换成合适的类型，来找出对象的类型。

```
Void GameEntity::OnCollision (GameEntity & entity)
{
   if (entity.IsType(GameEntity::PROJECTILE) ){
         GameProjectile & projectile = (GameProjectile &)entity;
         projectile.BlowUp();
}
// ...
}
```

3.3.2 C++风格的强制转换

看起来，使用C风格的强制转换可以把想要的任何东西转换成合乎心意的类型。那为什么还需要一个新的C++类型的强制转换呢？新类型的强制转换可以提供更好的控制强制转换过程，允许控制各种不同种类的强制转换。

C++里有4种不同类型的强制转换操作，取决于要强制转换的东西：static_cast、const_cast、reinterpret_cast和dynamic_cast。

C++强制转换操作与传统的强制转换相比，有一点语法上的细微差别。稍微有点冗长，而且在代码里有点突出。但是不要因为多敲几下键盘就烦了，它也比C风格的cast更能体现强制转换，新风格的cast是下面的格式：

```
static_cast<type>(expression)
```

下面的代码使用了C++风格的cast：

```
//C++ -style cast
float f2 = static_cast<float>(n) / 100;    //cast to a float
```

C++风格的强制转换其他的好处是，它们能更清晰的表明它们要干什么。程序员只要扫一眼这样的代码，就能立即知道一个强制转换的目的。是的，需要多敲几下键盘，但是确实值得。

1. static_cast

static_cast操作是比C风格的转换的更严格版本。它会通知编译器，尝试着在两种数据类型之间进行转换。和C风格的转换一样，它能在内置的数据类型间互相转换，甚至在有可能有精度损失的情况下也能转换。然而和C风格的转换不一样的是，static_cast只能在有联系的指针类型间进行转换。可以在继承体系中把指针转换来、转换去，但是不能转换成继承体系外的一种类型。

```
class A
{
};

class B : public A
{
};

//Unrelated to A and B
class C
{
};

A * a = new A;

// OK, B is a child of A
B *b = static_cast<B* > (a);

// Compile error. C is unrelated to A
C * c = static_cast<C*>(a);

// The old C cast would work just fine (but what would
// the program do?)
C * c = (C*)(a);
```

其余的不同在于，static_cast 不能实行对常量的改变。和 C 风格的强制转换一样，如果不能把一个类型强制转换成另外一个，那转换就会失败。

2. const_cast

const_cast 操作不能在不同的种类间转换。相反，它仅仅把一个它作用的表达式转换成常量。它可以使一个本来不是 const 类型的数据转换成 const 类型的，或者把 const 属性去掉。通常，没有必要把一个非 const 类型的数据转换成 const 类型的。这种转换会自动进行，因为这是一个不太严格的转换。换句话说，从 const 到非 const，只能通过强制转换来进行。被迫使用 const_cast 表明有东西没有正确地符合程序的设计。大多数使用 const_cast 的情况是由于调用老式的用 const 修饰的函数。如果发现调用自己的函数，竟然使用了 const_cast，那就赶紧打住，重新考虑一下设计吧。

3. reinterpret_cast

reinterpret_cast 操作符有着和 C 风格的强制转换同样的能力。它可以转化任何内置的数据类型为其他任何的数据类型，也可以转化任何指针类型为其他的类型。它甚至可以转化

内置的数据类型为指针，无须考虑类型安全或者常量的情形。reinterpret_cast 的结果就是实现的依赖性，它依靠每一个要转换对象的特定内存分布。使用 reinterpret_cast 的场合不多，仅在非常必要的情形下，其他类型的强制转换不能满足要求时才使用。

4. dynamic cast

在第 2 章里讨论了多继承的问题，曾简单地提到 dynamic_cast。所有其他的强制转换操作都是编译器在编译期处理的。至于转换的结果要么正常实现，要么导致一个编译错误。无论哪种情况，都不涉及到运行期开销。但是 dynamic_cast 有着显著的不同。dynamic_cast 仅能应用于指针或者引用，不支持内置数据类型。然而关键的不同在于，在运行期，会检查这个转换是否可能。它不仅仅像 static_cast 那样，检查转换前后的两个指针是否属于同一个继承树，它还要检查被指针引用的对象的实际类型，确定转换是否可行。如果可以，它返回一个新指针，甚至计算出为处理多继承的需要的必要的偏移量。如果这两个指针间不能转换，转换就会失败，此时返回空指针（NULL）。很明显，为了让 dynamic_cast 能正常工作，必须让编译器支持运行期类型信息（RTTI）。如果不希望编译器支持 RTTI，则必须找到 dynamic_cast 的替代方法。第 12 章将寻找这样的替代方案。

3.4 结 论

本章主要介绍了 C++的 3 个新概念：

- 关键字 const 允许标记一个变量为只读的。任何改变该变量的企图都会导致编译器错误。这个变量可以是简单的数据类型，也可以是一个复杂的类的对象。标记一个成员函数为 const 表明函数不能改变对象的状态，所以可以调用操作了 const 对象的 const 类型的成员函数。以指针或者引用的方式为函数传递对象作为参数（传进或者传出），const 作为这样的工具，还是很有用的。
- 引用是对象的另外一个名字。引用的表现和指针很像，但是也有明显的区别。总的来说，引用允许以一个较高的层次来操作对象，而不像指针那样，传递对象的实际内存地址。
- C++新引入的强制转换操作，使我们可以更明确要转换什么，怎样转换。还能提供一些编译期的检查。此外 dynamic_cast 为使用多继承的对象的转换引入了新的功能。

3.5 阅 读 建 议

关于 const 的使用，这里有很多不错的观点和指导：

Meyers, Scott, Effective C++, 2nd ed, Addison-Wesley, 1997.

Murray, Robert B., C++ Strategies and Tactics, Addison-Wesley, 1993.

Stroustrup，Bjarne, The C++ Programming Language, 3nd ed., Addison-Wesley, 1997.

更多关于C++风格的强制转换可阅读：

Meyers, Scott, More Effective C++, Addison-Wesley, 1995.

第4章 模 板

本章内容：

- 寻找通用代码
- 模板
- 使用模板的不足之处
- 使用模板的时机
- 模板专门化（高级话题）

有时候发现自己的工作中需要一遍又一遍重复地书写几乎完全一样的代码。因为这些代码需要的不仅仅是参数或者返回值不同，而且有时候是由于不同的类和数据类型，所以无法编写出一个通用的函数就实现这样的功能要求。C++中采用一个新的概念——模板来解决这个问题。运用模板，可以写出不依赖于特殊的数据类型的通用代码，于是在以后的开发中，不管是程序的任何部分，或者是对于不同的类，都可以重用这些代码。

4.1 寻找通用代码

在详细解释模板之前，先花点时间弄清楚为什么需要通用代码，看清楚真正的需求是什么，并通过一个例子来学习一下模板是怎样使用的。

我们会遇到这样一种简单的情况，花了很多时间，重复地使用曾经在过去的项目里面处理过的程序代码，比如链表。链表（list）在游戏程序中的应用很普遍：游戏世界里的所有游戏实体，需要渲染的场景节点，模型中的网格，在AI控制下将要执行的动作，甚至是游戏中所有玩家的姓名。现实的情况就是，一个完整的游戏会用到许多不同类型的链表。

那么该怎样实现这些链表呢？答案是有许多不同的方法。下面逐一查看这几种不同的方法。

4.1.1 方法一：在类中建链表

最直接的解决方法就是在类里面建链表，毕竟这没有什么难度。我们所做的就是加一个向前的指针（如果是双向链表就需要再加一个向后的指针），放入一些增加和删除元素的函数。

```
class GameEntity
{
public:
    // All GameEntity functions here
    GameEntity * GetNext();
    void RemoveFromList();
    void InsertAfter(GameEntity * pEntity);
private:
// GameEntity data
    GameEntity * m_pNext;
};
```

这样做真的能使程序简单吗？不一定。像许多糟糕的程序一样，这样做虽然能够实现功能，但是当它应用到一个规模比较大的程序里的时候，事情就变得很麻烦了（指的是链表的实现散布于很多类里面）。我们需要意识到的第一件事情是：即使我们是非常棒的程序员，我们也有可能犯错误。对于每一个与链表相关的类，几乎所有相同的代码我们都需要重写。这样很容易出现偶然忘记把指针指向 NULL，或者在删除链表的尾部数据时忘记检查为空了。没能正确地完成以上的任何一件事情（或者其他的很小的事情），都很可能导致程序崩溃。

假设我们很谨慎，第一遍写代码的时候全部正确，那么当修改程序的时候会出现什么情况呢？我们已经在不同的类里面写了自定义的链表代码，但是现在想修改程序，使用数据有效性检查，或者改为双向链表。设想一下，如何完成这么多类的修改工作。即使能完成所有的修改，而且没有任何遗漏（这实在是不太可能），做这样枯燥而乏味的工作保证肯定会让你发疯的。

如果你觉得这还可以忍受的话，就看看下面的一些其他麻烦。程序中不同的链表很可能有不同的形式。因为有可能其他程序员在草稿中就完成了很多链表的使用，而且他们对于链表的形式可能有不同的想法。在链表中你可能使用 GetNext()，但是他们可能使用 Next() 甚至其他更奇怪的方式。万一需要知道链表中有多少个元素，该怎么办？或许会使用一个 GetNumElements() 函数，因为在正常情况下调用该函数返回一个数值。但是 GetNumElements()在执行中调用函数 Count()，Count()函数会遍历列链表里面的元素，并统计元素数量。如果认为这是一个无足轻重的函数，或许你会吃惊不小。这里面的关键问题是没有一个统一的接口来遍历这些链表，而且你不可能清楚地知道那些函数底层究竟是怎样实现的。

由于这样的原因，你也不可能有一套在所有链表都通用的函数。对于一个标准链表来说，采用一个链表集执行下面的操作是可行的，诸如：给链表中的元素排序，移动链表中的元素，或者查找重复的元素。独立的命令集可以使每一个链表完成它个体需要的功能实现，当增加新的功能的时候，需要对不同类型的链表书写不同的命令，这将导致更多的代码重复和时间上的浪费。

4.1.2 方法二：使用宏

使用编译预处理的宏来解决上文中提到的问题。建立一组宏，这将使得向类中添加链表功能的工作变得简单。仅需要创建两个宏，接下来在使用它们的时候，所需要做的仅仅是添加 LIST_DECL（类名）到头文件，添加 LIST_IMPL（类名）到.cpp 文件中。

```
// In GameEntity.h
class GameEntity
{
public:
        // All GameEntity functions here
        LIST_DECL(GameEntity);
private:
        // GameEntity data
};
// In GameEntity.cpp
LIST_IMPL(GameEntity);
```

现在程序中的所有链表都采用了统一的形式，这样一来，当编写代码的时候，就能够清楚地知道系统将做什么。如果仍然需要修改操作链表的代码，只需要修改宏，这样程序中所有的类将立刻按照新的代码操作链表。

不幸的是，这种解决方法本身有不少麻烦。宏是出了名的难于开发、难于维护，尤其是难于调试。预处理程序将宏代码扩散到各处，所以当编译器执行到此处，并试图编译这段代码的时候，这段代码与手工写的其他代码没有什么区别。如果编译错误，所有包含这个宏的类都将被标记上，而没有准确的信息指明在宏的什么位置出了错。这将使调试大段的宏代码变得很烦人。

如果仅仅是这一个缺点的话，或许还可以接受这个办法在调试时存在的一点不方便。但是下面还有两个更基本的难题，这也是程序开发中任何一种宏都不能解决好的。

第一个难题是，还不能写出一个能完成任何类型链表功能的宏。尽管已经努力做到使所有类中的链表的形式统一，但是至今还是没有安全有效的方式在各种不同类型的链表中完成相同链表操作。在下一种解决方案（继承）中，还会提到这个问题。

第二个难题是，不能将宏任意放到需要的位置，以便它对于需要做的工作应付自如。到目前为止，已经成功地将链表功能添加到类里面。现在，控制链表里面的实体已经很容易了。但是，如果一个实体需要同时出现在几个链表中的时候该怎么办呢？举个例子来说，player 类可能想在视中保留一个实体链表以便它能在以后方便地选取目标。这样一来，是否需要再给每一个实体添加一个链表呢？或者说，如果希望一个实体同时出现在两个链表中，甚至三个链表中，该怎么办？很明显，不能把所有可能的这些数据结构添加到类中去。一个替代的办法是，创建这个链表成为独立的类，并且将链表的功能从其他的类中彻底拿

出来。在第四种解决方案中将看到关于这种方法的详细描述。

4.1.3 方法三：继承

不同于宏将链表功能添加到每个类中，我们可以采用创建一个以链表为成员的基类。这个类中包括了所有常见的链表数据成员，比如向前指针和向后指针，还有链表的一些常用功能，比如插入和删除操作。

任一希望获得链表功能的类，只需要继承这个链表类即可。而且这样做可以自动获得链表类的所有成员函数。即使这个类已经是一个继承了父类的子类，仍然可以使用多重继承从链表类中继承（如图 4.1 所示）。

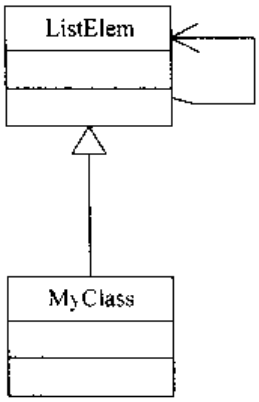


图 4.1 任一希望具有链表类属性的类都可以从 ListElem 类中继承

如果父类也是从链表类继承而来，那么问题将变得复杂了；我们的类体系将是一个钻石型的（参见第 2 章多重继承）。在这种情况下，虽然代码会按照预期设计的那样来执行（新的类可同时出现在几个链表内），仍面临很多二义性的问题，不得不区分代码正在进入哪个父链表。这是方法三处理问题时遇到的致命伤（关于多重继承的问题具体细节可以参看第 2 章）。

与宏解决方案不同，继承的方法对于调试没有任何困难。所有的链表代码都是常规代码，所以在调试时可以方便地看到，并能逐行进行检查，和调试其他普通代码一模一样。

这种解决方案的另一个优点是可以对链表成员采用多态。也就是说，既然有关链表的功能都继承自基类 ListElem，那么就可以在基类中添加指针功能和多种类型的 ListElem，这样就可以使程序中所有的链表获得这些功能。

继承的方法还是近乎完美的，尽管带来了多重继承的麻烦，但是如果就这一个麻烦的话，该方法仍然是可以备用的。继承的方法不能解决的最大问题是，它不能将链表功能和类本身分离开。前面也提到过，该方法不支持一个对象同时属于多个链表。因此，需要分离出链表的成员和链表的方法。

4.1.4 方法四：采用虚指针的容器链表

为了寻找到执行链表功能最好的解决方法，现在尝试把链表和它的内容彻底分开。可以把链表看成一个容器，并且选择需要的元素添加到容器链表中。

现在的问题是需要具有各种类的链表。可以为每一个想添加表的类建一个链表，但是这样一来进入了一个怪圈，最终回到方法一的路子上去。我们已经知道在类中建链表不是一个好的解决方法了。

如果使用链表而且不同类型的类需要继承同一个父类，我们可以这样做，于是链表中就包含了指向这个类的指针，虽然在实际的项目里面不大可能会这样。即使有一个游戏实体的根类，问题是网格、人工智能命令或者玩家（player）并不继承这个根类。除此之外，还需要整数链表、字符串链表、浮点实数链表，而且这些链表不是继承自同一个基类。

简单的解决方案是链表通过 void 类型的指针来处理，加进链表里面的任何东西都用一个 void 类型的指针指向它。当需要从链表里取数据的时候，就把该指针转换为需要的类型。链表代码除了在需要的地方复制它的成员变量之外，根本不需要对它们做其余的操作。这样只要添加的数据的指针与 void 类型的指针的大小一致就可以了，比如现在通行的 Windows 开发平台上 void 类型的指针的大小为 32 字节。至于它们具体是什么类型就不是那么重要了（如图 4.2 所示）。

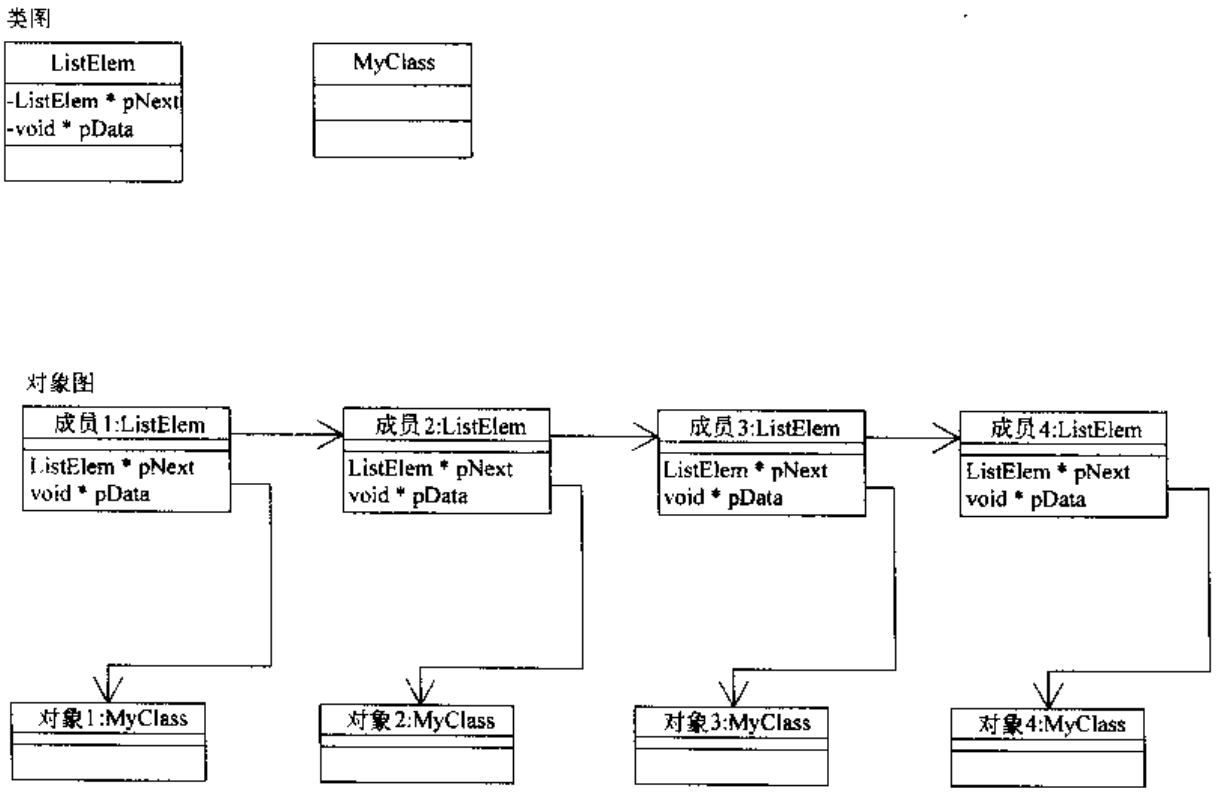


图 4.2 一个使用 void 类型的指针的链表可以用于任何类

我们现在担心的就是类型安全，这就要靠我们自己（靠代码）记住链表的成员变量是什么类型的。如果记错了，编译器并不能检查出来，而且将赋给成员一个错误的类型。幸运的话，这种错误会导致系统崩溃；如果很不幸，这种错误就会导致数据或者程序虽然能继续运行但是可能表现异常。要想查清楚这种性质的 BUG 是非常困难的，所以这也就成了该方法的主要缺点。

采用这种方法构建链表的优点之一是现在能够处理好链表类，而不是像过去那样处理链表的元素。采用这种方法还可以为链表添加一些全局操作和查询，诸如，获得元素的总数或者清空链表。

这种方法还有一个不足之处，虽然不太重要，但是会让人感到麻烦。我们已经将链表节点和链表包含的数据彻底分离了。这样做就等于是创建了新的对象并且把它加到链表将需要分配两份内存空间：一份分配给对象，另一份分配给指向这个对象的链表节点。像第一种解决方法那样，当链表的数据直接就是由对象组成的时只需要一份内存空间。在内存分配速度很慢或者有内存碎片问题的开发平台上，这个问题会比较突出（关于内存分配的更多知识和策略可参看第 7 章）。

为了找到更好的解决方法，有必要来看看 C++模板。接下来的内容将讲述模板的细节，然后重温这章中提到的问题，并使用模板解决它。

4.2 模　板

模板允许我们写一段代码而不必指定这段代码是某个类或者某种数据类型。这种方式书写的代码就很“通用”。当需要使用这些代码的时候，使用一个类来实例化该模板就可以了。模板有两种类型：类模板和函数模板，究竟是哪种类型的模板取决于被模板化的代码是哪种类型的。

4.2.1 类模板

从一个简单的例子开始，也许这个例子不值得用模板来实现，但是它可以很好地说明模板概念。例如游戏或者工具里面需要一个矩形类。矩形类有很多应用：窗口坐标、一些 GUI 组件的大小、视点的位置等。创建矩形类的一般方法是：

```
class Rect{
public:
    Rect(int px1, int py1, int px2, int py2){
        x1=px1; y1=py1; x2=px2;  y2=py2;}
    int GetWidth() {return x2-x1;}
    int GetHeight(){return y2-y1;}
```

```
    int x1;
    int y1;
    int x2;
    int y2;
};
```

以上的方法也许不是最好的类设计方法，而且它可能还需要添加函数才能更实用一些，但是现在它够用了。

目前看来，矩形类已经能够很好地为我们服务了，但是在以后项目开发的某些时候我们发现自己可能需要在不同的坐标系中使用一个矩形，而且它的坐标在0.0到1.0之间变化。因为重新定义的坐标是整数而不是浮点数，所以不能再次使用现有的类。解决的办法就是复制粘贴这段类代码，用float替换变量原来的数据类型int，生成类名为RectFloat的新类。

到这里，应该意识到任何需要复制粘贴的重复工作都是很麻烦的事情，是应该避免的。替代的方法是运用模板。使用模板之后，矩形类在实例化之前并不明确指明数据类型是整型还是浮点数。

```
template<class T>
class Rect{
public:
    Rect(T px1, T py1, T px2, T py2){
        x1=px1; y1=py1; x2=px2; y2=py2; }
    T GetWidth() {return x2-x1;}
    T GetHeight() {return y2-y1;}

    T x1;
    T y1;
    T x2;
    T y2;
};
```

从上例中可以看到，int的位置全部替换成了T，T就是建立模板用的类。可以用下面的代码构建数据类型是整型的矩形：

```
Rect<int> myIntRectangle (1, 10, 2, 20);
```

当编译器遇到这个定义时，它会搜索初始模板，用int替代模板代码中的所有T，并且迅速编译新的类。类似地，下面的代码可以构建数据类型是浮点数的矩形。

```
Rect<float> anotherRectangle;
```

有了模板，可以创建出多种多样的矩形来。这种方法的惟一局限性在于使用的类必须能够做减法，因为函数GetWidth()和GetHeight()都要求类可以支持减法运算。如果试图用

模板实例化的类不支持减法，我们可以捕获到编译错误。

值得一提的是当实例化模板为一个类的时候，编译器必须能够编译到模板的所有代码。这就意味着必须把有关模板的所有代码放在头文件里面；否则这段代码就是不可见的，也不会被编译（这将引起链接错误）。后面会看到，把模板代码放在头文件里面对于文件依赖性问题和编译时间都不利。

C++标准实际上规定了应避免将模板的执行代码放在头文件里面。标准允许模板的执行代码放在.cpp 文件里面，但是要用关键词 export 标明以便于编译器可以看到执行代码。这样的安排可以避免使用模板时再增加额外的依赖条件。不幸的是，现在还没有一个商用C++编译器能够实现这个功能。希望若干年后，可以看到支持这种功能的编译器出现。至于现在，惟一的选择就是把模板的所有代码全部放在头文件里。

4.2.2 函数模板

如果已经明白了类模板的概念，函数模板的概念就容易理解了。它们与类模板非常相似，只不过它们作用于函数而不是类。函数模板和类模板的主要区别是函数模板不需要明确的实例化。相反的，它们基于传来的参数类型自动创建。

下面是一个很简单的例子。函数 swap 包括两个变量，但是它没有被指定为任何一个类：

```
template<class T>
void swap(T & a, T & b){
     T tmp(a);
     a=b;
     b=tmp;
}
```

这个函数可以用于整型、浮点数、字符串以及任意有构造器和赋值运算符的类。传来的参数的类型决定了实例化的函数的类型。

```
int a=5;
int b=10;
swap(a , b);          //Integer version is instantiated

float fa=3.1416;
float fb=1.0;
swap(fa , fb);        //Float version is instantiated
```

注意一下，传给函数 swap 的两个参数是同一类型的。如果两个参数的数据类型不同会怎么样呢？我们做不到。因为这会导致编译错误。即使其中一个参数对数据类型有一点隐

敲的转化，这样做还是会导致编译错误。

```
// The following is illegal. Compile error.
swap(a , fb);
```

4.2.3 重温链表的例子：模板解决方案

借助于刚学会的关于模板的知识，再次研究这一章的前半部分提到的链表例子。这次用 C++模板克服前面提到的那些方法的局限。回顾一下，我们要完成的目标是书写一个具有以下特征的类：

- 这个类能够被广泛的应用于不同的类。
- 所有的链表代码始终在一个位置，不至于由于类的类型不同而复制代码。
- 所有的链表有标准的接口。
- 链表代码与类代码分开。

接下来能看到模板是怎么帮我们实现以上要求的。创建一个链表成员类型特定的链表类模板。同时创建两个不同的这样的类：一个是链表本身，一个是链表节点，两个类中都包括数据成员和链表指针。

```
template<class T>
class ListNode{
public:
    ListNode(T);
    T & GetData();
    ListNode* GetNext();
private:
    T m_data;
};

template<class T>
class List{
public:
   ListNode <T> * GetHead();
   void PushBack(T);
   //…
private:
   ListNode<T> * m_pHead;
};
```

像前面使用其他模板类一样使用这个链表类。

```
List<int> listofIntegers;
List<string> listofStrings;
```

现在可以毫不费力地创建处理各种类型数据的链表，而不需要把这些类直接和链表揉在一起。到目前为止，模板是解决这个问题的最好方法。

需要特别提醒的是，并不需要自己动手写一般数据结构的模板。C++标准模板库（STL）已经提供了大量不同数据类型和算法的模板，可以在程序里面自由使用。每一种数据结构是一个链表，可以和使用自己创建的链表一样使用它：

```
std::list<int> myListofIntegers;
```

在第 8 章和第 9 章会具体讨论 STL。

4.3 使用模板的不足之处

模板还远没有达到完善的程度，但它们的确是解决现有许多问题的最佳方法。了解模板的优缺点就和熟悉它们的句法结构一样重要。

4.3.1 复杂性

使用模板时最大的问题就是其复杂性。上一节中用模板解决了链表的实现问题，将其代码与采用第一种方法时的代码相比，哪一个更加易读呢？哪一个将来更便于修正和升级呢？答案是不言而喻的。

更糟的是，即使你检查了所有的小括号以及对一个属类 T 的引用，模板代码还是非常难以进行调试。我们承认，它虽然没有像预处理程序宏那样糟糕，但是也好不到哪儿去。当试图执行一段模板代码时，许多调试器将会陷入非常混乱的状态，一些调试器甚至会由于一个最细微的输入错误而出现非常难以理解的错误信息。

4.3.2 相关性

除了复杂性，模板还存在一些别的问题。正如前面介绍的，一个模板的所有代码需要存储于一个头文件中，从而保证当初始化一个模板时，这些代码对于编译器来说是“可见”的，但这增加了程序中不同的类之间的耦合。为了实现一个模板，任何一个含有模板的类将会自动包含所需的头文件。除此之外，即使仅仅产生一个很小的变化，也会明显地加长编译时间。这对一个大量使用模板的大型项目来说将是一个非常严重的问题，因为由于一个很小的变化而引起编译时间增加很长，这让人很难接受。

模板将会导致编译时间加长，且不依赖于类之间的耦合紧密程度。当初始化一个模板

时，编译器将产生一个新的类，并随之对其进行编译。幸运的是，这些额外的时间并不是一个很大的因素，并且对于多数项目可以忽略不计。

4.3.3 代码膨胀

代码膨胀是使用模板时经常遇到的另一个问题。当创建一个新类的链表时，编译器必须随之创建一个完整的新的链表类。这意味着需要复制所有的函数及所有的静态变量。当使用模板函数时，也同样需要做同样的工作，从而对于赋给函数的各个类型的参数，将产生新的函数代码。一般来说，代码膨胀问题还没有想象中的严重。和现在的游戏程序代码相比，它是微不足道的。只有在使用“模板的模板”的时候，才应该去考虑代码膨胀问题。因为使用“模板的模板”可能会导致模板的代码激增，编译器所产生的额外代码在整个项目中将会非常巨大。

如果代码膨胀成为项目中一个非常严重的问题，则需要尝试着将模板中的一些通用代码转移到一个标准函数中去。这样，当初始化一个模板时就不需要再重复这个函数了。然而这种做法通常是不可能的，也没有多大意义。这是因为，除非有大量的模板函数，否则很难将大量的通用代码转移出去。

4.3.4 编译器支持

最后，如果决定使用模板，必须考虑编译器和平台对它们的支持程度。模板最终被C++接受并成为标准，这需要时间。即使模板已经用了一段时间，它们仅仅作为被建议的方法，并不具备严格的标准。这就表明多数编译器不太支持模板的使用。幸运的是，多数编译器支持基本的模板，但是对于具有更高级特征的模板就不能提供有效的支持了，例如模板专门化。如果计划通过第三方程序库来大量使用模板，这一点就尤为重要了。

4.4 使用模板的时机

关于使用模板的建议和对继承的建议一样，那就是要谨慎地去使用它们。要充分考虑现在（以及将来）开发小组成员的专业水平，并据此来确定模板的使用。如果最后期限已迫在眉睫，而又必须在一个大型项目中去理清一大堆乱七八糟且设计蹩脚的模板，这是非常困难的，也是令人烦恼的事情。这时就希望一个更为简单的办法。

记住，如果为了利用高级的C++而去使用模板是毫无意义的。模板只是一个工具，它能够使过程简化并节省时间。但是如果现在为了节省几个小时，却会导致多花费几天的时间，此时使用模板并不能算是一个好办法。

也就是说，总会有这样的时候，当想将一段代码应用于许多完全不相关的类中去，同

时还希望这些类之间继续保持这种完全不相关性。这时，C++模板将会是一个很好的解决方法，这与本章中前面列出的例子很相似。

一种最适合使用模板的就是容器类，它是包含许多不同类的对象的数据结构。幸运的是，许多容器类已经在 STL 中写好了（详见第 8 章）。但是，还有一些特殊的容器类是 STL 不能使用的，比如树容器、一些特殊的具有优先权的队列，以及其他一些依赖于特定环境的容器类。在这样的情况下使用模板将是一个很好的方法。

或许还希望看一看 boost 库。在很大程度上来讲，它是一套大量使用模板并试图对 STL 进行拓展和补充的 C++程序库。如果在 STL 中找不到所需要的，那么就去 boost 库中去检查。如果仍旧找不到的话，再决定自己去编写。

尽量用一种类似于 STL 的形式去编写模板代码是一个很好的建议。boost 库就是一个很好的例子。用类似于 STL 的形式去编写模板代码将可以更加容易地与其余代码相结合，使其他程序员更加方便地去熟悉 STL。另外，它甚至允许和一些 STL 算法、迭代程序以及一些容器类互相作用。其他的一些适合使用模板的有管理器/工厂类（一些专门创建及跟踪对象的类）、资源加载代码、单件模式（singletons，某类仅有惟一实例）以及对象的序列化操作等。

4.5 模板专门化（高级话题）

默认地，在一个模板中创建的所有新的类都将会完全一样，只是它们被应用于不同类型的类中。在编写通用代码时，遇到的问题是，我们几乎一点不知道这些通用代码将会运行在什么样的数据类型上。对于较大的对象，我们明确地希望能尽可能地避免去复制它们，因此增加一些指针来避免复制好像是一个好办法。另一方面，一个很小的对象甚至是指针自身都能够很容易地被随意复制。所以使用指针来管理一个只有 32bit 的对象似乎是一个非常没有必要的举措。

模板专门化允许在一个特定的类或一组类的模板中增加一些自定义形式。这样，就能够为打算经常使用的类做一些优化，例如指针和字符串等。优化可以表现为更高效地执行或者占用更少的内存，这取决于实际需要。

4.5.1 完全模板专门化

第一种类型的专门化是完全模板专门化，它允许将一个模板的自定义形式提供给一个特定的类。

再来回顾一下本章前面介绍的一个简单的模板。它为每一种数据类型提供了链表的一个实现，通过充实代码链表的实现变得非常高效，接着在游戏中的每一部分都要使用它。到项目的最后，意识到有很多包含较小单元的链表。特别地，在每个地方都要使用游戏实体句柄的链表。

游戏实体句柄是一个很小的类，它只处理整型数且不具备实际功能及继承类型，所以这种类的对象仅有 32bit。对每个实体句柄分配一个链表节点是非常浪费的。一个链表节点需要两个指针，这将使需要处理的数据规模扩大 3 倍，而且由于是动态分配，需要占用额外的存储空间，从而浪费更多的内存（关于内存分配以及利用内存池系统来解决该问题的具体细节可参阅第 7 章，内存分配）。

现在，通过使用模板专门化来改善一下情况。可以自定义链表模板的形式，该链表模板的单元存储在一个连续的内存块中。在这个链表中插入和删除单元是很费事的，但是由于该游戏实体的规模是很小的，同减少的内存使用相比，这个代价很小。

下面的代码不能够替换上一个模板，但是它能够对其进行补充。同时它必须在要专门化的模板声明之后才能出现，而不能出现在该模板声明之前。专门化模板所执行的每个函数都会使总模板中的默认函数无效。

```
template<>
class List<GameEntityHandle>{
public:
    ListNode<GameEntityHandle> * GetHead();
    Void PushBack();

private:
    GameEntityHandle * m_pData;
};
```

当然，随着模板的声明，将能够执行那些专门管理游戏实体元素的表，这些游戏实体元素是存储在一个连续的内存块中的。在这种情况下，还需要为链表节点 ListNode <GameEntityHandle>提供专门化，这样它就能够和链表的自定义形式共同运行了。

4.5.2 部分模板专门化

假定这样一种情况，如果使用的链表不属于同一类型，但其单元却是共用的，例如（这种情况在使用面向对象设计及多态时经常发生）多数链表含有不同类型对象的指针。指针很小，和一个游戏实体对象差不多大，所以它们比较适合专门化的链表操作。对于这种情况，将使用第二种类型的专门化，那就是部分模板专门化。下面的代码利用部分模板专门化为一个包含不同类型指针的表创建一个模板。

```
template<class T>
class List<T *>{
public:
    ListNode<T*> * GetHead();
```

```
    void PushBack();

private:
    T * m_pData;
};
```

当初始化一个新类型的表时，将会根据初始化该模板时所使用的类来选择特定的模板。

```
// Normal templated list is used
List<Matrix4x4> matrixList;

// Fully-specialized list, because it stores GameEntityHandles
List<GameEntityHandle> handleList;

// Partially-specialized list, because it stores pointers
List<Matrix4x4 *> matrixPtrList;
```

4.6 结　论

本章介绍了模板的概念。模板使我们在编写代码时能够不依赖于特定的数据类型，并且编译器能够在程序需要的时候对它们进行实例化。有两大类模板：类模板（应用于整个类）和函数模板（应用于单个函数）。模板的主要优点如下：

（1）在不同的位置不需要复制代码；

（2）具有数据类型安全性；

（3）无需从一个通用基础类中进行继承。

然而，模板还具有以下缺点：

（1）代码更加复杂且难以维护和调试；

（2）包含过多的相关性，增加了编译时间；

（3）代码规模增大；

（4）编译器支持不够完全。

了解模板的优缺点以及何时使用模板是开发项目时的一个重要方面。最后介绍了模板专门化的概念。模板专门化使得我们能够为一个特定类或一组类自定义模板。利用这一特征，可以为特定的数据类型编写更优形式的模板。

4.7 阅读建议

下列文献里有对模板的一般性介绍和模板用法的讨论：

Murray, Robert B., C++ Strategies and Tactics, Addison-Wesley, 1993.

Stroustrup, Bjarne, The C++ Programming Language, 3rd ed., Addison-Wesley, 1997.

关于模板的更深入更全面的介绍可以在下面的书中找到：

Vandevoorde, David, and Nicolai M.Josuttis, C++ Templates: The Complete Guide, Addison-Wesley, 2003.

下面的这本书对于模板使用有一些先进的延伸性的思考：

Alexandrescu, Andrei, Modern C++ Design, Addison-Wesley, 2001.

第5章 异常处理

本章将讲述：

- 错误的处理
- 异常的使用
- 异常的保护代码
- 异常的开销分析
- 异常的使用时机

C++为处理错误提供了新的技术：异常处理。异常处理允许写出简单而健壮的代码，用于正确处理各种错误信息和意想不到的情况。本章将要介绍的内容包括：使用异常处理的原因，怎样在程序中有效地使用异常功能，在使用异常功能时系统的开销，尤其是它们如何影响程序的执行效率。在学习的过程中，将结合一些游戏开发的例子来介绍异常处理的使用。

5.1 错误的处理

自从有了计算机以来，程序员就不得不面对错误，并对错误进行处理。但一直以来，或者花费大量的时间和精力来考虑错误，或者忽略错误，从来都没有过一个理想的解决办法。这一节中，介绍了处理错误的常见方法，同时引出了C++异常的概念。

5.1.1 忽略错误

第一种方式是当遇到错误时忽略错误。这种方式或许比较可笑，但这正是大多数程序处理错误的办法，或者说是大多数程序员确实就是这样处理错误的。当然了，可以检查文件是否已经打开了，但又有谁会检查了printf函数的返回值呢？

大多数程序将能够通过这种方式顺序的执行。如果没有意想不到的事情发生，一切都会运行得很好。然而，一旦有其他的情况发生，事情会变得很糟，程序可能会崩溃。当内存不足时会怎样？如果桌面的显示设定了8位色会怎样？如果磁盘空间不足时会怎样？当游戏运行时没有足够的磁盘空间或者游戏运行过程中光盘取出，又会怎样呢？

忽略错误或许是一种很好的方法，其简单快捷，或许甚至是适合于内部开发的工具软件（这主要依赖于你所在的公司对内部产品品质的期望）。但很显然，这并不是所期望的，

我们不能把这样的程序出售给用户，所以需要去寻求其他的解决方法。

5.1.2 错误码

我们所使用的一种由来已久的方法，就是检查函数的返回值。每一个失败的函数调用都会返回一个错误码，或者至少有一个布尔变量指出这一调用是否成功，调用函数的代码再检测返回值，并恰当地处理任何失败的调用。

理论上来讲，这种方法或许是切实可行的。但在实际当中，当应用于一个完整的设计时，这种方式就显得不足了。一个函数本来很简单，仅有两行代码，使用了这种处理方法后，变得面目全非。也许发展成有着30行代码，但是被复杂的if-then-else结构搞的异常混乱。这种方法，不仅仅是使得程序面目全非，更严重的是，它使得我们难以辨别出函数到底在干什么。关于错误处理的代码搞乱了函数的真实目的。

下面的这些代码在载入数据的过程中存在漏洞。其中，第一段程序没有进行任何形式的错误检测，第二段进行了错误检测，比较一下，哪一段更容易阅读。

```
void Mesh::Load(Stream stream)
{
    ParseHeader(stream);
    ParseFlags(stream);
    ParseVertices(stream);
    ParseFaces(stream);
}

int Mesh::Load(Stream stream)
{
    int errCode = OK;
    errCode = ParseHeader(stream);
    if (errCode !=OK){
        FreeHeader();
        return errCode;
    }
    errCode = ParseFlags(stream);
    if (errCode !=OK){
        FreeHeader();
        return errCode;
    }
    errCode = ParseVertices(stream);
    if (errCode !=OK){
        FreeHeader();
```

```
        FreeVertices();
        return errCode;
    }
    errCode = ParseFaces(stream);
    if (errCode !=OK){
        FreeHeader();
        return errCode;
    }
    errCode = ParseHeader(stream);
    if (errCode !=OK){
        FreeHeader();
        FreeVertices();
        FreeFaces();
        return errCode;
    }
    return errCode;
}
```

源代码表示的是它本身。但没有让错误的代码很容易地脱离出来，仍然有很多的问题存在于每个函数所产生的错误代码中。错误检测代码不仅难看和混乱，而且是非常浪费的。如果要调用的每一个函数都有 if 条件语句存在的话，那游戏的总体效果会显著下降，这里指的是游戏的性能。

于是产生了相关的问题，应该在程序的什么地方保留错误检测代码。错误代码是单独的成为一个巨大的文件，每一个程序员都可以#include 的；或者是引擎的每一个子系统都有它自己的代码。这些程序怎样能够将错误的数值（比如一个数字，-134）代码转化为可读性更强的字符串？肯定会有更好的办法的。

5.1.3 使用断言

一个合理的处理错误的方法就是什么都不做，意外的情况是计算机在遇到错误时自己崩溃了。可以通过使用断言函数（assert）使程序在遇到错误时停下来（这一功能将在第 16 章进行详细的介绍）。程序结束之后，至少可以知道产生错误的文件和错误在文件中的位置，以及一些描述信息。这一功能虽然比什么都不做要强一些，但还是留下了很多的工作要做。

5.1.4 setjmp 和 longjmp

做系统开发的 C 程序员通常都很熟悉 setjmp 和 longjmp 这两个函数。setjmp 允许我们

在程序代码中（以栈的形式）保存一段可以将来调用的代码，longjmp可以用来恢复那个特定的地址和栈的状态。当有错误时，可以假设通过调用 longjmp 功能来处理错误。但不幸的是，除了不够灵活之外，这种方式在C++中并不是非常地适用。longjmp可以打开堆栈中保存的代码，但是不能够处理其中的东西，从而使得存放在栈中的对象永远不会被释放，析构函数没有得到执行。这意味着内存泄漏或者有些资源永远不能被释放。如果想立刻退出程序的话，这样或许是比较简便的方法；但是，如果想要忽略错误并进一步执行程序的话，这样做或许会导致意想不到的结果，程序的资源将被耗尽，最终程序会崩溃。

5.1.5 C++异常

现在介绍C++的异常。到目前为止，所有的已经提出的错误处理机制都存在着明显的缺陷。在了解C++异常的确切的语法之前，先来介绍一下C++异常通常是怎样处理的，以及为什么要用它来替代以前的错误处理方式。

1. 异常的工作方式

异常功能是遵照以下方式工作的：每当程序遇到非正常情况时，它就会抛出一个异常。该异常使得程序跳到最近的一个异常处理模块，如果该模块里没有这个异常的处理，程序会将栈中的代码取出（正确的销毁栈中的每一部分），并且进入父函数（调用自己的函数）继续寻找异常处理模块。在找到相应的异常处理模块之前，这种迭代方式会一直进行下去。如果迭代到栈的顶部（函数的调用是通过进栈出栈来进行的）时，还没有发现有对应的异常处理模块，这时程序自动调用默认的异常处理代码，然后程序本身会停下来。

在一个异常处理模块中，可以做任何想做的事。异常处理模块可以报告错误，并尝试对错误产生的问题进行修改，或者忽略错误。通过异常处理模块，还可以尝试解决不同的问题，这主要取决于异常产生的具体原因（比如说，或许对于文件损坏的异常和拿零作为除数的异常，在处理上有些不一样）。一旦异常处理模块完成了工作，程序将恢复正常，从这里开始继续执行下去，而不是回到产生异常的地方继续执行。

2. 优点

异常处理的第一个优点在于它不会产生凌乱的代码。一个函数在使用了异常处理之后，看起来和之前没有任何的错误检测时是一样的。整洁的源代码不仅仅是好看一些，而且更加容易读懂和易于处理。

异常是非常灵活的，对不同的错误它可以采取不同的处理方式。有时可以对产生的错误进行修改和恢复；有时可以像断言函数那样终止程序。

异常还可以将原始的错误经过简化以后传递给较高层的程序代码。通常来说，在程序的文件IO底层代码发生运行错误时，我们希望通过各种途径，运用一系列的调用，在GUI界面上反映出所产生的错误。要想通过返回值实现这一功能，所有在这一过程中与其有关的功能都需要准备返回错误码。而在异常处理功能中，所产生的错误将会自动的被传递到

最近的 try-catch 模块，并且在这些异常对象中还包含了很多除错误代码之外的其他相关的有用信息。

在讨论返回代码的时候，还有一种情况没有讨论，那就是构造函数。构造函数是没有返回值的，从而当错误反馈机制是以返回错误代码为基础的时候，情况就变得非常麻烦了。虽然可以在对象上设一个变量来标记失败，但是如果这样的话，调用的代码必须在操作对象之前首先检查该变量。作为另外一种选择，可以不对构造器产生的错误进行任何处理，但这样会限制它的使用。有一个习惯用的方法叫做“在初始化时获得资源”，这一工作主要依赖于对象在其创建的时候对其自身的初始化，这样可以少一个单独的步骤来检测。这使得我们总是可以对程序进行恰当的初始化，而不必不停地对程序进行检测。

顺便提一下，析构函数的情况也是一样的，由于对象正在被销毁，析构函数除了能够判断出程序是否失败之外，也做不了什么。但在对象被销毁以后，任何针对该对象的操作都不能再进行了。无论如何，仍然希望在释放对象的过程中，至少能够反映出发生了什么样的错误。

你可能在想，这些功能到底先进在哪里？异常处理使得我们能够发现在构造函数内部的错误，很快就可以看到这项功能有多么完善。

最后，再提些相关的东西。异常处理功能避免了在整个项目范围内建立一个需要不断维护的错误代码表。在这里，不需要将错误的代码转换为人能够读懂的错误信息。通常情况下，异常本身就包括了错误描述信息，这使得调试过程更加容易。

5.2 异常的使用

在前面的一节中，给出了在使用异常时的一些优势。当然它也有不足的地方，这样的情况将会在后面的章节里面遇到。现在先来看一下怎样在程序中使用异常。

5.2.1 概述

在 C++中使用异常的语法是非常简单的。当程序代码需要产生异常的时候，程序就通过关键字 throw 抛出异常。程序的控制系统会在关键字 catch 的引导下跳到最近的异常处理模块。在可能产生异常的代码的周围，每一个 try 模块后面都要跟有一个 catch 模块。下面是一小段简单的代码，其中包括了异常的产生和对异常的处理：

```
void f1(){
    printf ("Start function f. \n");
    throw 1;
    printf ("End function f. \n");
}
```

```
main() {
    printf ("Start main. \n");
    try {
        printf ("About to call f. \n");
        f();
        printf ("After calling f. \n");
    }
    catch ( ... ){
        printf ("Handling exception. \n");
    }
    printf ("Ending main. \n");
}
```

这段程序的执行结果是：

```
Start main.
About to call f.
Start function f.
Handling exception.
Ending main.
```

注意，异常一旦产生，函数f()中余下的语句就跳过不再执行了，虽然这些语句的代码就在try模块内部，函数会跳到catch模块中正常执行。

5.2.2 产生异常

正如在前一节中所看到的一样，产生的异常是以产生一个相应的关键字（throw）作为标识的。这种方式使得C++的异常使用非常灵活，可以将任何形式的对象作为异常来处理。

在前面的例子中，只是抛出1这个异常（throw 1），并进行了处理，因为没有关注所产生的异常属于哪一种类型，只是关注抛出一个异常。但在大多数时候，还是希望得到一些关于异常的更清楚的信息。如果在程序代码中，不同的部分因为不同的原因产生了不同的错误，当追踪到异常的时候，怎么能够知道产生异常的原因呢？

为了解决这一问题，可以创建一些异常对象，这些对象中包含有所需要的所有信息。当遇到异常时，可以随时抛出这些对象，并将它们传送到catch语句中。下面是一个简单的例子：

```
class MyException{
public:
    MyException ( const char*pTxt) : pReason(pTxt){ };
```

```
        const char * pReason;
};
```

为了使用这个对象，需要抛出一个带有这种类型异常的对象。下面这一段代码试图从一段数据流中读取顶点信息，如果由于文件太短而不能读取足够的数据的话，它就会产生异常：

```
void Mesh::ParseVertices (Stream stream){
    for (int i=0; i<m_numVerts; ++i) {
        VertexData data;
        int nRead = stream.Read(data, sizeof(VertexData));
        if (nRead <sizeof(VertexData)
            throw MyException("File too short");
        Vertex vertex (data);
        m_verts.push_back(vertex);
    }
}
```

在后面将会看到，这个对象是怎样被传递到 catch 结构中的。可以将这种形式应用于多种不同的异常处理。

在程序中，产生异常的原因是多种多样的，这就需要针对不同的情况写不同类型的异常代码。例如，我们能够分类，有一类与文件 IO 有关的异常，与图形硬件有关的异常，与数学运算有关的异常等。这是非常正确的，因为需要用不同的方式来处理不同类型的异常。例如，与图形硬件有关的异常要比与文件有关的异常严重。

为了实现这些功能，可以生成一些有层次结构的 C++异常类。通过将这些类排列在不同层次中，不但可以达到共享某些代码的目的，而且可以利用类的层次结构的优势来处理 catch 语句中的异常。图 5.1 中就是一个可能的异常层次结构。

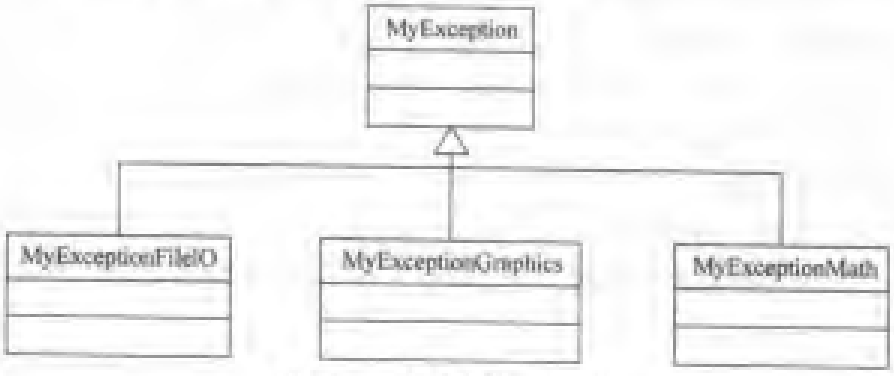


图 5.1　一个异常的层次结构

C++已经针对不同的情况将异常组织成了一些标准的层次结构。其中包括一些通用的

类型如 logic_error 和 runtime_error，特殊的类型如 out_of_range、bad_alloc，以及 overflow_error。处理这些异常的功能，在C++的标准类库中都有描述，所以需要了解怎样正确地处理这些异常。

5.2.3 捕获异常

到目前为止，只看到了使用 catch(...)语句捕获异常的过程。也就是说，只需要捕获到所有的异常，而不考虑异常的类型。

1. 捕获某种类型的异常

通常情况下，捕获类型明确的异常是最常用的。可以通过对 catch()参数的设定来实现对某种特定类型异常的捕获。下面的一段代码显示了怎样在自己编写的程序中捕获文件 IO 异常。

```
void Mesh::Load (const char * filename){
    try{
        Stream stream (filename);
        Load (stream);
    }
    catch (MyExceptionFileIO & e) {
        // Do something with the exception here
    }
}
```

既然假设在文件中产生一个新的 stream 或者是从 stream 中载入一个 mesh 会产生异常，于是在代码的周围设置了 try 结构，在代码的后面跟有 catch 结构。如果在这些代码中产生了 ExceptionFileIO 型的异常，代码会捕获到并对它进行处理。

2. 捕获多种类型的异常

如果产生了异常，但这个异常与异常处理程序要求的类型不同会是什么情况呢？我们的代码处理不了这个异常，于是系统会启动传递链功能，将异常传递下去，直到找到可以处理这一类型异常的处理代码；或者是到达传递链的末尾，在这种情况下，会调用预先设定的 terminate 函数，这一功能会终止程序。

我们宁愿处理不同类型的异常，也不愿将程序中断。我们可以设计多捕获模块，类似一个 switch 结构，每一部分处理一种异常。在上面的例子中增加一个处理数学异常的异常，结构：

```
void Mesh::Load ( const char * filename){
    try{
```

```
        Stream stream (filename);
        Load (stream);
    }
    catch (MyExceptionFileIO & e) {
        // Do something with the exception here
    }
    catch (MyExceptionMath & e) {
        // Deal with it here
    }
}
```

如果必须对每一种想要处理的异常都写 catch 语句的话，是非常令人讨厌的。我们常常会遇到大量不同类型的异常，这些异常都需要我们来写处理代码，这种情况是首先应当避免的。

还记得是怎样在层次结构中组织异常的吗？现在同样可以利用这种组织方式。一个 catch 语句可以处理由参数进行标识的一类异常，或者是由这种类型所派生出的任何类型的异常。也就是说，不必对异常处理程序做很清楚的说明，可以为 MyException 这种异常类型写处理程序，它可以处理所有 MyException 派生出的所有类型的异常。

继续来看前一个例子，在这个例子中并不专注于捕捉数学类异常，而是仅仅希望保证在 Load 结构中不出现异常。通过使用异常的层次结构可以用一种更好的方法来实现这一功能。

```
void Mesh::Load (const char * filename){
    try {
        Stream stream (filename);
        Load (stream);
    }
    catch (MyExceptionFileIO & e){
        //Do something with the exception here
    }
    catch (MyException & e){
        // An exception that was not file IO-relaced was thrown.
        //Deal with it here.
    }
}
```

在 catch 结构中，这种顺序是非常重要的。程序从头开始运行，一直运行到最后。一旦发现了异常，就会调用最近的异常进行处理，余下的部分就被忽略。

如果希望 Load 功能是绝对可靠的，确保不会漏掉任何异常的话，甚至可以捕获到不是由基本异常类所产生的异常。这就需要在程序的末尾再加上另一个 catch 语句。

```
viod Mesh::Load(const char * filename){
    try {
        Stream stream (filename);
        Load (stream);
    }
    catch (MyExceptionFileIO & e) {
        //Do something with the exception here
    }
    catch (MyException & e) {
        //An exception that was not file IO related
        //was thrown. Deal with it here.)
    catch (...){
        //A different exception was thrown.
    }
}
```

3. 重新抛出一个异常

迄今为止，当遇到异常的时候，总是假定已经对它进行了处理，程序照常执行下去；也有可能发现了异常，并尝试着去处理它，可是处理不了，希望程序的其他部分来处理；也有可能发现了异常，经过进一步详细的分析，发现它并不是想要处理的内容。

当异常处理程序决定不对某个异常进行完全的处理的时候，它可以重新抛出该异常。为了实现这一点，需要使用关键字 throw，不需要任何参数，在 catch 语句内部，会产生一个相同的异常，让下一个 try-catch 语句来进行处理。

在下面的这个例子中，仅仅处理了由于错误数据所引起的异常。其他的异常，包括其他文件 IO 异常，会由程序的其他部分进行处理。

```
void Mesh::Load (Stream stream)
{
    try {
        ParseHeader(stream);
        ParseFlags(stream);
        ParseVertices(stream);
        ParseFaces(stream);
    }
    catch (MyExceptionFileIO & e) {
        if (e.IsDataCorrupt()){
            //Handle corrupt data here
    }
```

```
    else {
            throw; // Throw the same exception again so it
            //can be handled somewhere else
    }
  }
}
```

5.3 异常的保护代码

在程序中使用异常，所要知道的不仅仅是怎样调用 throw 函数和构造一个 try-catch 语句块。为了有效的处理异常，需要编写代码来正确的处理异常，不能出现内存泄漏和资源耗尽的情况。

5.3.1 资源的获得

在 C++中，所谓资源是指代码可以访问的东西，但这些内容需要进行明确的释放，以便系统的其他部分可以利用。这些资源可以是内存的形式，文件句柄的形式，或者是传统上在游戏的开发中称做资源的东西，例如纹理、几何模型、声音等。

1. 问题

可能存在的问题是：如果获得了一个资源，而同时又产生了错误，怎样释放该资源。这并不是一个新问题，在所使用的所有的错误处理程序中都会遇到这样的问题。异常功能的使用，使这一问题变得更加突出，因为在它周围没有什么可以被容易引用。

下列的例子以最简单的方式说明了这一问题。这个例子假定读取文件中来创建一个纹理，它打开文件，得到纹理的长和宽，给纹理分配内存，并为纹理读入数据。到目前为止，还完全没有做任何错误检测。

```
Texture  * CreateTexture(const char * filename) {
    FILE * fin = fopen(filename, "rb");
    TextureHeader info;
    ReadTextureHeader(fin,info);
    Texture * pTexture = new Texture(info.with, info.height,
                info.colorDepth);
    ReadTextureData( fin, pTexture);
    fclose (fin);
}
```

这一节会通过多种方式来重写该函数（使用检查返回值、使用异常等方式），现在仔

细看一下这个函数。这一函数中包含有多少资源呢？我们并不是很清楚 ReadTextureHeader 和 ReadTextureData 这两个函数做什么，假设它们并没有创建资源。CreateTexture 具有两个资源，一个是文件句柄，另外一个是在这一函数中间产生的纹理对象。当然，只有文件句柄被释放了。纹理已经被传递到了调用 CreateTexture 的函数里了。

2. 使用返回值

如果在错误处理功能中采用了检查函数的返回值，前面的一段程序会是什么情况呢？函数大概会是下面的样子：

```
Texture * CreateTexture(const char * filename) {
    FILE * fin = fopen(filename, "rb");
    if (fin == NULL)
        return NULL;

    TextureHeader info;
    if (!ReadTextureHeader(fin, info)){
        fclose (fin);
        return NULL;
    }

    Texture * pTexture = new Texture (info.width, info.height, info.
colorDepth);
    if (pTexture == NULL) {
        fclose (fin);
        return NULL;
    }

    if (!ReadTextureData(fin, pTexture)){
        delete pTexture;
        fclose (fin);
        return NULL;
    }

    fclose(fin);
    return pTexture;
}
```

可以看到在程序当中到处都是重复的代码，在程序中增加了许多步骤使得程序更容易产生错误。如果改变习惯使用 goto 语句的话，程序会相对地清晰一点：

```
Texture * CreateTexture(const char * filename) {
    bool bSuccess = false;
    Texture  * pTexture = NULL;
    TextureHeader info;
    FILE * fin = fopen ( filename, "rb");
    if(fin = =NULL)
       goto cleanup;
    if (!ReadTextureHeader(fin, info))
       goto cleanup;
    pTexture = new Texture(info.width, info.height, info.colorDepth);
    if (pTexture = = NULL)
       goto cleanup;
    if(!ReadTextureData(fin,pTexture))
       goto cleanup;
    if(!ReadTextureData(fin, pTexture))
       goto Cleanup;
    bSuccess = true;

  cleanup:
    if (!bSuccess) {
    delete pTexture;
    pTexture = NULL;
   }
   fclose (fin);
   return pTexture;
}
```

这样做仅仅是稍微好了一点。至少在使用了 goto 语句之后，限制了直接跳到程序结束的情况，所以程序仍然是可读性很高的。

3. 使用异常

当使用异常功能的时候，在有错误的地方也不需要使用 goto 语句了。这样，源程序代码就不会因为进行错误检测而发生混乱了。

```
Texture * CreateTexture(const char*filename){
   FILE * fin = NULL;
   Texture * pTexture = NULL;
   try{
       fin = fopen(filename, "rb");
       TextureHeader info;
       ReadTextureHeader(fin, info);
```

```
        pTexture = new Texture(info.with, info.height, info.colorDepth);
        ReadTextureData(fin, pTexture);
    }
    catch (...){
        delete pTexture;
        pTexture = NULL;
    }
    fclose (fin);
    return pTexture;
}
```

迄今为止，这当然是所有办法中最简便的了。当然了，如果有更简洁的方法的话，它当然是显得大了一些。此外，如果想把该异常交给调用自己的函数处理的话，该怎么办呢？那就需要重新写 try-catch 语句块，释放掉相应的资源，重新用 throw 抛出异常就可以了。

4. 通过初始化获取资源

一个更加简洁的获得资源的途径是使用“通过初始化获得资源”这一概念。当产生了异常，并且退出了当前函数时，所有在这一函数中产生的对象都会销毁。在这种情况下，由于 TextureHeader 类型中 info 对象在栈当中，所以被销毁了。但是变量 fin 和 pTexture 仅仅是指针，所以它们确实没有析构函数。我们希望的是它们的析构函数能够释放它们所使用的资源。

在这个文件句柄的例子当中，将创造一个简单的类，将文件处理包装起来。当对象生成的时候，文件被打开；当对象被销毁时，文件被关闭。现在所要做的仅仅是在栈中创建一种该类型的对象，一旦离开栈，文件的资源就将被释放。

对于另一个资源，纹理指针，有一种相对更简单一些的解决办法，即使用 auto_ptr 类，这种办法可以应用于所有的指针。auto_ptr 类是 C++类库的标准类，与刚才处理文件句柄的类很相似，但它是应用于指针的。只要 auto_ptr 类被销毁，它包含的指针就会被销毁。除此之外，它还可以被引用，像一个普通的指针一样使用它。可以在这个例子中看到，这个新类是怎样自动释放资源的。

```
auto_ptr<Texture>CreateTexture(const char * filename) {
    FilePtr fin (filename, "rb");
    TextureHeader info;
    ReadTextureHeader (fin, info);
    auto_ptr<Texture> pTexture = new Texture (info.width, info.height,
                                              info.colorDepth);
    ReadTextureData(fin, pTexture);
    return pTexture;
}
```

这样就好多了。注意，在这个函数里没有做任何异常处理，而是让调用它的函数去处理。虽然，所有的资源都已经被正确地释放了。

但是除了 auto_ptr 类之外，还有其他的类可供选择，其中的一些类更适合一些特定的情形，这主要依赖于是否想共享类的所有权、提供引用计数器，或者是其他的功能。Boost 库提供了另外一些不同的智能指针，可以用来代替 auto_ptr。

同样，通过使用返回错误代码来处理错误的技术虽然不值得提倡，但这样做确实能够提高代码的可读性，允许函数立刻返回，并且清楚地知道所有的资源会被正确地释放掉。

最后，如果存在于对象里的资源能自动释放，那资源就有可能被释放多次。事实上这种情况可能是存在的。如果异常确实存在，并且几乎是不可能预知的，或许在特定的环境下，泄漏一些内存反而是好的，只要不滥用就可以了。

5.3.2 构造函数

构造函数通常有很多种形式。正如前面所说的，构造函数没有返回值，所以构造函数不可能通过返回一个返回值来指出构造函数执行失败。

由于它不依赖任何返回值，所以此时使用异常就是一个很好的选择。不过，这也带来了一个新的问题，先来看一下下列这些代码：

```
class Bitmap {
public:
    Bitmap (int width, int height) {
        m_pData = new (width*height);
        //...
    }
    ~Bitmap () { delete m_pData; }
private:
    byte * m_pData;
};
```

这是一个很直观的类。当构造了一个类之后，系统就会专门分配给它足够的内存，如果对象被析构了，系统就会自动释放内存。目前看来，代码还是很简洁的。

在系统给 m_pData 分配了内存以后，如果产生了一个异常，会发生什么情况呢？就像前面所提到的一样，栈会被打开，程序会转移到最近的一个 try-catch 模块。但是，刚才分配成功的内存会怎么样呢？内存会发生泄漏，这个资源还没有释放。

原因在于这一对象的析构函数没有被调用。毕竟，因为产生了异常而导致析构函数失效，所以不可能再去调用析构函数了。然而，在构造函数里生成的对象会被自动销毁。不幸的是，使用了一个“笨指针”（不是“聪明的”或“正常的”）。所以，尽管指针被销毁了，但这一指针所指向的内存地址仍然被占用着。

可以采用一种异常安全的方式，通过使用 auto_ptr 指针创造一个同样功能的构造函数，从而使得 auto_ptr 对象被销毁以后，内存就会被释放，还有另外一个好处，不需要在析构函数里释放内存，内存会自动地被释放。

```
class Bitmap {
public:
    Bitmap (int width, int height):
        m_pData (new (width * height)){
        //...
    }
private:
    auto_ptr<byte> m_Data;
};
```

5.3.3 析构函数

对于异常处理来说，析构函数不像构造函数有那样多的问题。毕竟是在销毁一个对象，不会再销毁之后又对它进行存取。

需要牢记在心的原则是，当析构函数响应了另外一个异常以后就不允许再产生异常了。也就是说，在程序中的某个地方产生了异常，析构函数可能会被调用。但是，如果析构函数执行的过程中又产生了异常，系统会怎样做呢？是忽略旧的，处理新的？还是顺序处理它们？C++不允许这种情况发生，但如果确实发生了这种情况，程序就会调用 terminate 功能，终止程序的执行。

如果还是担心这种情况发生的话，可以把可能在 try-catch 模块析构器中产生异常的调用包装起来，放到 try-catch 语句块里，该 try-catch 语句块捕获所有的异常，要么忽略掉，要么采取一些方法去处理。

5.4 异常的开销分析

异常处理的细节完全留给了编写编译器的程序员，因此没有绝对的标准可以讨论执行处理异常的性能开销。但另一方面，程序出现异常是怎么处理的，具体的实现方法也是很重要的。对于游戏程序而言，这一点就更为重要。和其他 C++的话题一样，在这一节中，把所期望的目标作为一个通用的标准，比如在平台上所产生的结果，或者在异常处理时看到了一些汇编代码。不同的情况怎么衡量呢？

首先，必须区分两类完全不同的情况，一种是没有异常产生的情况，另一种是产生了异常的情况。

当产生了一个异常，就会接连发生一系列的事：程序开始寻找最近的异常处理模块，

销毁进程中产生的对象，转向执行 catch 结构。虽然这一操作比较浪费时间，但并不介意。

异常本来就应该表示不正常。异常并不是用来指出一个游戏中的怪物是否被杀掉，而是用来让我们知道内存是否被耗尽，或 DVD 是否从驱动器中拿走了。这是真正的异常环境，不会发生或者发生的很少。我们并不介意产生和捕获异常的快慢，并可能会临时停止程序并告诉用户发生了什么问题。当然对于回滚栈信息（对应的情况是，出现的异常自己处理不了，需要上一级的调用来处理）这样的事情而花费 100ms，我们还支付得起。

我们所关注的是当没有异常产生的时候，程序性能是否确实受到了影响。另外一个有意思的问题是，使用异常处理功能是否需要更多的内存。于是，产生了另外两个相关的问题。

执行异常处理有两种主要的方式。第一种方式同时也是最简单的方式，仅需要很少的额外的内存。只是在它执行相应的 try-catch 语句的时候，会稍微有一点点性能开销。第二种方式则没有任何性能开销，只是这种方式需要较多的内存来有效地处理异常。最糟糕的情况是不知道用了哪种实现方式，而不是改变编译器。如果遇到了这种情况，查阅编译器的文档找出究竟使用了哪一种方式。

关于异常的实现，还有另外一个重要的方面需要考虑。那就是 try-catch 语句的使用频率以及是否存在其他的方法来替代它。如果 try-catch 语句在程序代码中出现的频率很低，或者是出现在不重要的地方，例如资源载入的过程，那程序的性能开销是不会太显著的。另一方面，如果在每一个渲染帧里，try-catch 结构都执行了成千上万次，那总体的性能可能就会比较显著了。

同样，在评估异常处理性能的时候，同样要考虑有没有其他的可以替代它的方法。不能把它们与完全没有错误检查的代码比较。如果确信没有错误发生的话，那么就不要使用异常，从而也就不会在异常的处理上浪费时间了。但大多数的商业工具软件和游戏都需要一定水平的容错性，于是就不得不通过其他一些方法来处理错误。使用异常需要进行进栈出栈操作，这和许多个 if 语句嵌套混乱不堪的代码结构相比，就显得合理多了。

另一方面，如果编译器实现异常处理是通过耗费内存来换取速度的话，这种方式的缺点可能就在于你是否能够提供足够多的内存了，特别是对内存非常有限的游戏终端（游戏机）来说。在准备使用异常或忽略异常之前，应该先做一些测试。

5.5 异常的使用时机

到现在，已经知道怎么使用异常处理了，怎样改变异常处理程序满足我们的要求，它的性能开销情况怎样。在实际应用中，应该怎么使用异常呢？

首先，重申一下前面的观点，异常确实是异常，永远不要忘记这一点。使用这一功能的一个好的原则是：在所有可能发生错误的地方使用异常，而且这错误不可预料。当发生了异常，使用者就要注意了。有一些能够较好的解释异常使用情况的例子，如：数据文件被损坏，硬件运行错误，格式错误的媒体文件，与互联网的连接突然断开等。必须通过某种方式来处理这些情况，但在正常使用的情况下它们是不会发生的。

也就是说，异常处理并不能够完全取代函数错误返回值检测的方法。在许多情况下，检查返回值还算是最好的解决办法。例如：试图打开一个在先前的游戏软件中保存过的文件，但文件名拼错了。试图打开一个并不存在的文件，这时返回一个错误值就是一种最合理也很有效的途径。有时，再执行一项操作，并且想要知道操作是否成功。显然，执行失败并不是所期望的结果，这时返回一个错误的值是较好的，同时也是最简单的解决办法。

两种办法可以共同存在而不产生矛盾，它们分别适用于不同的场合。程序的不同部分使用什么样的错误处理需要编写文档来说明，这是非常重要的。不希望别的程序员在返回错误代码的周围又写下很多的 try-catch 语句块。

迄今为止，存在一种争论，是仅在开发工具软件时使用异常处理呢？还是在游戏开发中使用异常呢？对于游戏终端会怎么样呢？对于两种情况来说，答案或许都是肯定的。只要异常出现的几率很小，那使用异常处理能够省掉很多凌乱的代码，也不需要在调试程序的时候小心观察 if 语句究竟执行哪个分支了。

5.6 结 论

在引入异常之前，看到在程序中处理错误的相关的方法，包括：忽略错误，利用返回值代码，或者是使用 C 核心的 setjmp 和 longjmp 功能。所有这些方式都存在明显的缺陷。

异常可以用很简洁的方式报告错误的存在。它不需要在代码中写很多的 if 语句，而且非常可靠，因为异常是不容易被忽略的（不像返回代码容易被忽略）。不幸的是，写异常安全代码不是太轻松，特别是在构造函数中。

接着看到了不同的异常处理具有不同的性能，并且需要不同的内存。然而，如果异常被用在了错误可能出现的地方，那么在游戏的开发中使用它们，其带来的性能开销会很小，可以被忽略。

5.7 阅 读 建 议

这里有很多有关异常处理的参考书目。这都是很好的参考书，其中一些书给出了很不错的建议，包括使用异常处理的合适场合，怎样书写异常安全代码。

Stroustrup, Bjarne, The C++ Programming Language, 3rd ed., Addison-Wesley, 1997.

Murray, Robert B., C++ Strategies and Tactics, Addison-Wesley, 1993.

Dattatri, Kayshav, C++ Effective Object-Oriented Software Construction, Prentice Hall, 2000.

Sutter, Herb, Exceptional C++, Addison-Wesley, 2000.

Boost 库中有很多的智能指针可以用来有效地进行异常处理。

C++ Boost Smart Pointer Library http://www.boost.org/libs/smart_ptr/index.htm.

第2部分　性能和内存



一个游戏的成功，不外乎这么几个要素：激发兴趣的交互性，让人屏住呼吸的视觉表现，以及超前的音效。现在一个游戏能否成功取决于它的运行表现。如果是一台配置不好的老机器，游戏运行起来每秒种也就 5 帧的话，那还不算太糟，但绝对不会太受欢迎。如果游戏运行 10min 后由于内存泄漏无法继续玩下去的话，那确定会有大批愤怒的顾客要求退货的。

内部开发工具同样也有性能上的要求。如果加载关卡编辑器需要好几分钟，或者如果卷轴的时候能够感觉到画面反应迟钝、很慢，那游戏设计者的工作效率以及最终产品的质量肯定会受到影响。

C++是非常复杂的程序开发语言。这有两面性：由于它的复杂性，可以实现其他语言仅能梦想实现的东西；相反的一面，它太难掌握了，学习周期长，还有编写程序的时候很容易出错。功能强大才有高可靠性。

本书的第 2 部分主要针对 C++的一些技巧和高级话题：如何避免性能方面易犯的错误，如何加速程序，如何有效地管理内存，怎样使用更容易、更有效的高级数据结构和算法。

第6章 性 能

本章内容：

- 性能和优化
- 函数类型
- 函数内联
- 函数开销更多的方面
- 避免复制
- 构造函数和析构函数
- 数据缓存与内存对齐（高级话题）

有人认为 C++太复杂了。读了上一章的内容后，也许读者已经开始陷入 C++的技术细节。不必否认 C++确实是很复杂的程序语言。但是幸运的是，玻璃杯的水并不仅是半空的，同时它也是半满的。

C++是复杂，但是正是由于它的复杂性，才带来了程序的高性能。把 C++和其他语言相比，如 Java，一般认为它们很容易学，不是那么复杂，但是这些语言对于程序的性能却没有提供太多的可控制性。对于游戏编程来说，可控性显得特别重要。其他语言也许能提供相似的性能，但是一般不能应用到 PC 之外的其他平台。

在本章将看到 C++性能方面容易出现的问题，以及 C++如何有效地解决这些问题。特别地，能看到不同类型的函数调用，它们涉及到的性能开销，以及如何使这种开销给程序带来的影响最小。本章也会详尽地研究一些隐藏在 C++背后的性能损失，在源代码里面看不到的恰好又隐藏在代码背后的东西，比如临时对象或者构造函数和析构函数的开销。

理解隐藏在 C++背后的情况，并牢记其中的一些，就可以使 C++的复杂性变得驯服一些，于是获得 C++能提供的性能就是手到擒来的事了。

6.1 性能和优化

性能不是一切。专门单开一章来讲述性能问题，可能看起来有点让人感到奇怪，但这是千真万确的。从某种程度上来说，把所有的精力集中在程序的一小部分，运用本书讲到的技巧，努力使程序运行得更快一点。榨干硬件所能提供的最大能力，这是很让人愉快的事。有时候这可能是必需的，但是大多数情况下不是这样的，没有必要费那劲。图 6.1 的例子说明了一个很好地使用了图形加速卡的典型的 PC 游戏的性能情况。

图 6.1 可以说明几个有意思的事情。第一，时间多花在显卡上，也就是渲染上。所有的 C++的优化都不能使显卡更快一些。与此相反，需要再看看其他使游戏的图形更有效率的办法，比如重新排列顶点数据，减少状态切换的次数等。

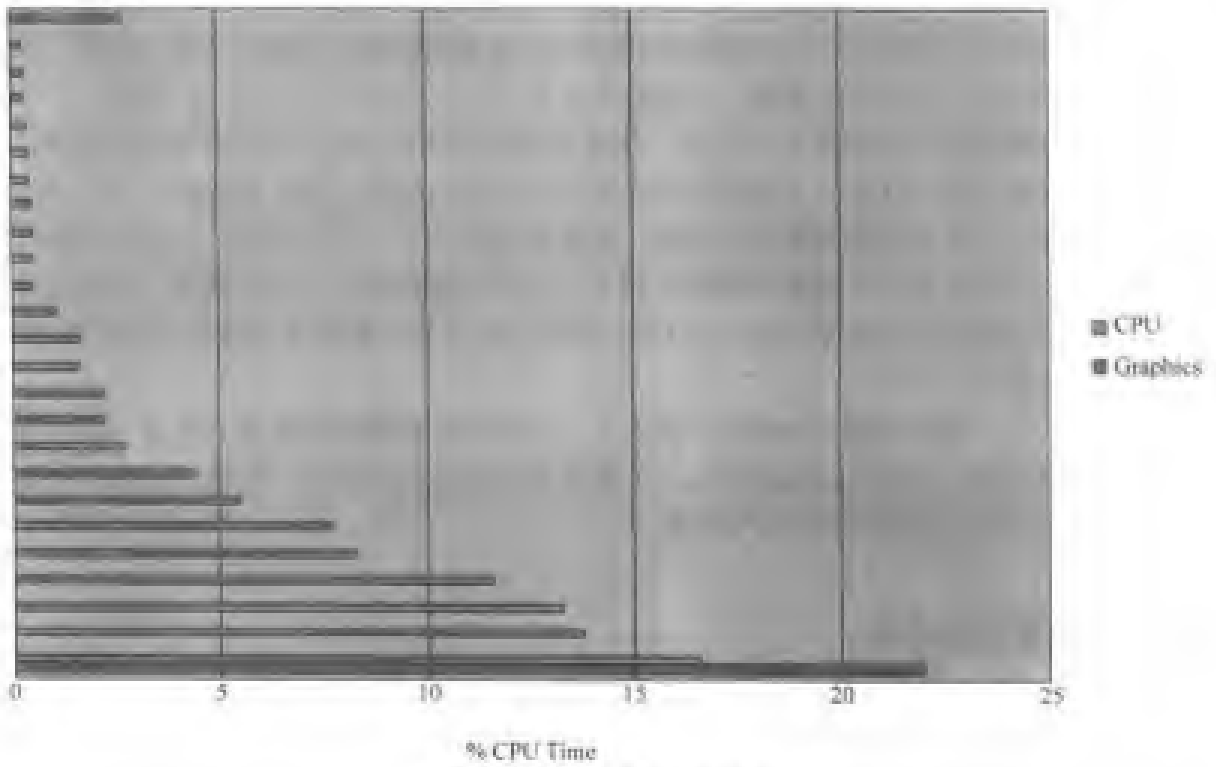


图 6.1 典型 PC 游戏性能图

从 CPU 的角度来说，这有一个有意思的规律：90%的代码执行时间花费在大约 10~12 个不同的函数上。如果费了力气所优化的是其他的函数（不是瓶颈的函数），把这个函数的性能提高了 2 倍。帧速率肯定也不会有明显的提高，那我们的努力就有点浪费了。

更糟糕的是，即使优化了性能最差的 10 个函数，也许没有得到性能上的任何提升，因为 CPU 可能不得不等待显卡的操作（显卡渲染太费时间了，是整体性能的瓶颈）。将来，就像看到的很多特殊的硬件一样（为不同的用途而专门设计），并行处理会变得更为重要。现在大多数的游戏终端已经是这样了，它们除了主 CPU 之外，还有一些专用芯片。

好像这样还不够可怕似的，考虑一个例子吧。这个例子虽然是杜撰的，但是距离真实不会太远，从中看看不合理的优化所带来的可怕后果。程序员 Pete 确定要优化 AI 寻路函数，所以他花了一整天的时间，使出浑身解数，最后终于使得寻路函数的性能提升了 20%。他很得意自己的高明，高兴得手舞足蹈。他很得意地把代码签入库，然后回家好好休息了一下。

第二天早上，程序员 Pete 重新 build 了整个游戏，他注意到速度并没有明显提升。当查看帧速率（FPS）的时候，他发现确实提高了一点：大约 2%。这并不奇怪，因为 AI 的寻路算法占了 CPU 使用率的 10%；所以总体的性能提升并没有他期望的那么显著。但是任何小的改进都是可取的，所以 Pete 把注意力转到了别处。

过了一个星期，有人要求程序员 Pete 为飞行类物体增加寻路算法。到目前为止，他的程序都是处理地面上的物体的，他觉得没有问题。他拿出上周那个晚上通宵赶制出的优化后的代码，他发现自己不能很清晰地加上飞行类的物体。代码优化的太厉害了，几乎无法改动了，不仅如此，Pete 甚至不敢动这些代码了。因为它实在是太难理解，每一行是什么意思都搞不明白了。对这样的代码做任何的改动，都会弄坏寻路算法，而且项目有一个里程碑（milestone），从时间上来说，也没有几天了。

这个故事的结局是程序员 Pete 取消了他所有的优化代码，在原有代码的基础上加上了飞行物体的寻路支持。这是个不错的结局。其他的可能结局是，Pete 很顽固，强行在优化后的代码上加上了飞行类物体的寻路支持，但是他却花费了三天的时间，而且这样的代码从未正常工作过，于是在里程碑的期限没有拿出东西，最终整个项目被取消。这个故事的寓意说明，在一个不合适的时间段优化代码，或者在项目的开始就过多地考虑优化的事情，会带来很多问题。

没有人否认，性能问题是重要的。事实上，这个问题对于游戏来说，远比其他类型的软件开发重要得多。需要很谨慎地对待，但是也要选择合适的时机。性能问题有两个显著的阶段：程序开发过程中和开发收尾阶段。

6.1.1 程序开发过程中

程序开发过程中主要的目标是避免做很没有效率的事情。不要太关注确切的循环次数，是否应用了汇编代码等。要尽量避免做一些会使程序其他部分慢下来的事情。本章指出了在程序的开发阶段中会出现的一些事。一个很好的例子就是小心一点，在这个阶段，如果没有必要，就不要作对象的复制。

有的时候，优化什么，哪些需要留到后面优化，做出这样的决定，不是太明确。这大多取决于经验，项目小组里的专家的意见，还有自己的能力。一个笨方法就是，如果还不能确定是否进行一个特定的优化，那就先不管它。

也是这个阶段，牢记算法的性能非常重要。不要太关注于查询算法是否优化，但是与此同时，要避免在一帧里在一个很大的列表里多次查找同一个元素。相反，要用一个变量把查找结果存起来，当需要的时候，直接使用这个结果就可以了。或者，如果知道你的游戏世界里有很多单元和对象，要避免为了查找一个元素就遍历整个列表，而要使用特定的可以依据位置来快速定位元素的数据结构。

6.1.2 开发收尾阶段

收尾阶段并不是指准备要发行游戏了，它其实可以是开发一个特定的子系统（比如图形渲染子系统）快要完成的时候，或者是觉得已经足够了，不准备再琢磨寻路算法了的时候。

从这一点来说，可以使用一些技巧来优化。将用程序的可读性和可维护性来换取高性能。如果想在将来的项目里复用这些代码，就需要多考虑到底怎么优化它。如果代码封装的很好，那好好优化一下会非常值得。要不然，就得在代码里为所有的优化写好注释，以备将来之用。翻出你过去写的代码，这些代码优化的不是很清楚，它运行的很快，但是你不知道它做了什么。这样的事情经常发生。

牢记这个阶段的优化通常不会导致明显的性能的提升。很大的收获来自于前一阶段（开发阶段）所做的决策：算法的选择，避免不必要的计算等。优化代码本身会给我们带来散布在各处的一点点的性能提升，从总的情况来看，程序的性能也许能提高10%~15%，但是别期望更多了。

本章的观点是什么？本章不是讲述如何操作数据位或者用汇编代码替换C++代码的，不是讲述选择最好的算法来解决手边的问题的，也不是讲述避免做重复或者不必要的工作的。本章主要是讲述在日常的C++应用中，如何避免可能带来的性能问题；还有在设计代码的时候或者实现代码的时候，需要牢记的一些事情。

6.2 函数类型

过去在使用其他编程语言的时候也用过函数调用，也谈到过每一次函数调用都有性能问题，在这里又提出这样的问题，主要有两个原因：

- 除了一般的函数调用，C++还有类的成员函数调用、类的静态函数调用和虚函数调用。了解每一种函数调用类型有助于提高程序的性能。
- C++鼓励使用面向对象的方法，这会更多的涉及函数调用而不像面向过程的程序那样。因此，函数调用的开销会比C语言里的更引人关注。

在C++里，函数调用分成4种类型：全局函数、类的静态函数、非虚的成员函数以及虚函数。从性能的观点虚函数又分为两类：单继承下的虚函数和多继承下的虚函数。

6.2.1 全局函数

这和在C语言里遇到的函数调用一样，例如：

```
int nHitPoints = GetHitPoints ( gameUnit );
```

这种类型的函数调用被解析成一个特定的内存地址的调用，该地址是在链接的时候计算出来的。因此惟一的性能开销来自于内存地址的计算以及对代码缓存和CPU管道线的影响。在现在的大多数平台上，这样的开销基本是可以忽略的。可以人为创建这样的环境，在这样的环境之下，一个全局函数的调用的性能开销变得很显著。然而这样的环境在真实的开发环境下几乎不可能碰得到，尤其是函数被正确地处理成内联的情形（这正是下一节要讲述的内容）。由于全局函数是最简单，也是运行速度最快的函数类型，所以把它作为

分析比较其他类型的函数调用的底线。

通常，一个需要遵循的很好的规则就是：忘记性能开销问题。养成自由地使用函数的好习惯有很多好处，能够看出任何微小的性能缺陷。要这样使用函数：

- 鼓励使用较小的函数，行数不要太多；
- 书写可读性强的代码，因为每一个函数只有一个惟一的目的，通过函数的名字就可以直接了解函数的大致情况；
- 书写可维护性强的代码，这样当解决一个规模较大的问题时，可能有很多函数，但只要改动其中之一就可以了；
- 鼓励代码的复用，因为很多问题可能有一些共同的步骤。

过多使用函数的主要缺点在于，很难弄清程序执行的流程，尤其是当这些函数分散在不同的文件里的时候，更难跟踪程序的流程。拥有一个好的工具，比如一个 IDE（Integrated Development Environment，集成开发环境）或者代码浏览器来管理这些代码，在这样的场合下就显得非常有用了。如果在某一段代码里，函数调用的开销变得显著了，借助这些工具就很容易把几个相关的函数合并为一个，以消除性能的降低。

6.2.2 类的静态函数

除了仅仅局限在一个特定的类以内，类的静态函数很像全局函数。它们和成员函数的不同在于，它们和这个类的对象没有一点关联，仅和类本身有关系。

调用类的静态函数，编译器在处理的时候和全局函数同样对待。它们仅仅在代码的外观上有细微的差别，因为需要用范围定界符来表明它们属于哪个类。

```
//Class static function
int numUnits = GameUnit::GetNumUnits();
```

当说到性能和用法的时候，同样的注解也适用于一般的静态函数。类的静态成员函数提供了一种手段，可以把一组相关的函数集结到一个类下面，这样做的目的是为了帮助程序员能够更好地理解代码。

6.2.3 非虚的成员函数

成员函数是一个特定的对象的函数。这里仅讨论没有被标记为虚函数的成员函数。

```
GameUnit gameUnit;
int nHitPoints = gameUnit.GetHitPoints();
```

这种类型的函数很像前两种类型。以前，要调用的函数的地址是在链接的时候确定的，因为调用的对象的类型也是编译的时候确定的（但是虚函数的情形下将会改变）。

惟一的区别在于成员函数引用了一个特定的对象。有一个隐藏的参数：this，this 指向了那个被调用的对象。从内部来讲，上面函数的调用将会转换为下面的样子：

```
GameUnit gameUnit;
int nHitPoints = __GameUnitClass__GetHitPoints( &gameUnit );
```

多出来的下划线（__）仅仅是编译器为了识别成员函数属于哪个类而改的名字。这就叫 name mangling，这和上面的例子相比，少了一点吸引力。

和全局函数相比，成员函数有其余的性能开销吗？惟一的多余开销是参数的传递。大多数情况下，这和全局函数没有什么差别。所以可以自由使用成员函数。本章后面的内容将会详细讲述参数传递所带来的性能问题。

6.2.4 单继承下的虚函数

虚函数有可能开销很大。关于虚函数的具体实现，不管单继承还是多继承，可参见第 1 章和第 2 章。通常情况下，在使用多态的对象的情况下才使用虚函数。继续上面关于 GameUnit 类的例子，现在有了一个 GameUnit 类的引用，但是它指向一个派生的类。RunAI() 是一个在 GameUnit 类里声明的虚函数。

```
GameUnit * pGameUnit = new SomeGameUnit ;
pGameUnit->RunAI();
```

这个例子里，RunAI()的调用将触发虚函数的机制，而这会调用虚函数表从而导致额外的开销（参见第 1 章）。下面的等价代码展示了编译器是怎样解析上面的函数调用的：

```
GameUnit * pGameUnit = new SomeGameUnit ;
(pGameUnit->vptr[3])(pGameUnit);
```

这个例子里，RunAI()是虚函数表的第三个虚函数，所以编译器通过虚函数表来调用它，准确地说是通过 this->vptr[3]()来调用的。注意那个额外的参数（其指向所作用的对象）是怎样传递进去的，就像非虚的成员函数那样。

额外的性能开销会由于平台的差异而有所不同。除非在每一帧里它被调用成千上万次，否则它的性能问题完全可以不用考虑。就像在第 1 章里看到的那样，如果真正需要一个虚函数提供的功能，那它就会该多快就多快。

实际上，这不太准确。任何类型的虚函数机制都需要一定的迂回手段才能进行函数的

调用，就是由于迂回手段发生在不同的地方，才导致虚函数机制的不同。使用 C++继承，迂回手段发生在要调用的这个类的虚函数表的索引上。每一个类有一个虚函数表，可以保存在内存的任何地方，也许离对象本身很远。意思是说，访问虚函数表可能会导致缓存的数据丢失。这样一种情况是常常出现的：为很多不同的类调用了虚函数，工作的平台恰好数据缓存又很小。每当缓存的数据丢失的时候，工作的平台内存存取数据就很慢。比如有些游戏终端（游戏机），这个问题变得就更复杂了。这种情况下，为每一个对象保存自己的函数指针可能就会快一点。这是一种典型的内存空间平衡，但是在一些非常特殊的情况下，会导致一些差异。

由于虚函数表的缓存数据丢失而导致的对程序性能的影响，也是特别难以消除的。程序的反应迟钝反映出程序什么地方存在着问题，表面上很简单的循环都会耗费很长时间。对它哪怕只要做一些很小的改动，都会是危机四伏。如果碰上了相似的情形，也由于调用虚函数使程序运行开销昂贵，则应当好好考查这种可能性。

你也许会疑惑，继承体系的大小能对虚函数调用的性能产生什么后果呢？如果继承层次很深的话，是不是意味着性能一定很糟糕？单继承的情况下，答案是否定的。每一个类都有自己的虚函数表，这与它是怎样派生出来的没有关系。

最后还有一种情况，虚函数调用的开销并不比一个一般的静态函数调用开销大多少。这发生在虚函数直接被对象而不是对象的指针或者引用调用。C++仅支持引用上的多态，不是针对对象本身。编译器能够解析编译后的最终地址。这绕开了运行期的虚函数执行机制。

接下来的例子，虽然 RunAI 是一个虚函数，它也是像任何非虚的成员函数那样被调用，这是由于它是被对象（而不是对象的指针或引用）直接调用的缘故。

```
GameUnit gameUnit;
gameUnit.RunAI();
```

另一方面，下面的代码展示了同样的函数通过对象的指针被调用的情况。由于编译器没有办法知道运行期对象的真实种类，该函数调用将被编译器使用虚函数查表的办法来调用。

```
GameUnit * pGameUnit;
pGameUnit->RunAI();
```

6.2.5 多继承下的虚函数

第 2 章里介绍了多继承是如何实现的，某个使用多继承的类的虚函数表是如何创建的，父类的虚函数表是如何追加到自己的虚函数表后面的等。当调用一个虚函数的时候，

像单继承的情况一样，编译器需要为虚函数编制索引。多继承独特的一点在于，取决于该虚函数是从哪个父类继承而来，虚函数表的指针需要调整一个偏移量到正确的部分。这个偏移量的调整只带来很小的性能开销。

其他的一些方面就和单继承的一样了，甚至由于虚函数查表导致的数据缓存丢失这种情况。尽管自己的虚函数表是由它的父类的虚函数表混合而成的，在内存里它们还是位于邻近的位置上。

广泛使用多继承的类，比如一个有着复杂继承体系的继承树，其“叶子类”可能会有一个非常大的虚函数表，这是由于它可能从它所有的父类继承了一堆函数。这样的话，当调用同一个对象的很多不同的虚函数的情况下，缓存数据丢失将变得更严重，因为这个虚函数表太庞大了，所以它可能导致缓存数据丢失及总体性能的下降。

6.3 函数内联

函数内联是这样一项技术，在某些特定的场合，它可以显著地减少函数调用的开支。为了更有效地使用它，必须理解它的工作机制，理解它适用的场合。

6.3.1 内联的必要性

考虑以下的代码：

```
class GameUnit {
public:
   bool IsActive() const;
   //...
private:
   bool m_bActive;
};

bool GameUnit::IsActive() const{
        return m_bActive;
}

   //...

   if ( gameUnit.IsActive() ) {
        //...
}
```

为了使用 if 语句判断，必须调用函数 GameUnit::IsActive()，这个函数调用的开销和一般常规的函数调用一样。

和在前一部分看到的那样，这个函数调用的开销是很小的。但是函数本身太微不足道，所以为了调用函数，而付出了微小的代价，这看起来就是浪费。此外，如果需要在每一帧都调用这个函数成千上万次，那就会突然产生额外的开销，导致帧速率的显著下降。

可以把 m_bActive 声明为 public 类型的成员变量，于是就可以直接访问它了。这样确实能避免额外的性能开销。不过从长远的观点来看，就可能得不偿失了，因为它会导致更大的麻烦。有的时候需要强制限定不要让类拥有 public 类型的成员变量，因为可能会有代码直接访问这样的成员变量，当类的实现需要修改的时候，可就麻烦了，因为跟踪哪些特定的变量被外部访问了是一件很困难的事情，而且几乎不可能限定某个变量是只读类型的。

这个办法会带来更大的问题，当函数不是返回一个成员变量的值的情况下，怎么办？如果例子中的函数 IsActive()是这么定义的：

```
bool IsActive() const { return m_bActive && m_bRunning };
```

事情开始变得有点不样。如果去掉该函数，让代码的其他部分直接访问这个成员变量，那意味着，不得不暴露 m_bActive 和 m_bRunning 这两个成员变量。当代码的其他部分想要检查一个游戏对象是否被激活的时候，不得不检查这两个成员变量。如果后来决定函数 IsActive()应该这样的时候，又怎么办？

```
bool IsActive() const { return (m_bActive && m_bRunning)|| m_bSelected };
```

想象一下，到处修改代码，这绝对是一场恶梦。还是找点其他办法吧。

6.3.2 内联

可以使用内联函数。内联有着和直接存取成员变量一样的性能，也没有一点副作用。所要做的仅仅是用关键字 inline 标记一个函数。在调用的地点，编译器会小心地把该内联函数调用去掉，并嵌入函数的具体内容。刚才的例子如果用函数内联来实现大概是这样：

```
class GameUnit {
public:
   bool isActive() const;
   //...
private:
   bool m_bActive;
};

inline bool GameUnit::isActive() const{
         return m_bActive;
```

```
}

// ...

if ( gameUnit.IsActive() ) {
    // ...
}
```

从函数调用这个方面来看，这和过去的调用是一样的。但在内部实现上，编译器会替换函数调用为函数的具体内容，如下所示：

```
if ( gameUnit.m_bActive ) {
// ...
}
```

函数调用的所有开销都没有了。对于小的但是被频繁调用的函数来说，把它标记为内联函数可以使程序性能得到极大提高。

使用函数内联很直观，但仍然需要注意以下几点：第一就是要内联的函数必须在头文件里定义。这是因为编译器只能“看到”我们告诉它要编译的那个文件里包含（include）的头文件。如果编译器想替换函数体里的具体内容，它必须能够访问它。事实上，编译器还没有聪明到自己可以打开.cpp文件的程度。

有两个办法声明一个内联函数，并且在头文件里定义。函数的定义可以在声明之后立即进行。

```
class GameUnit {
    inline bool IsActive() const { return m_bActive };
    // ...
};
```

或者函数的定义可以放到函数声明的后面，像在例子中看到的那样。这样代码显得整洁一些，这适合函数的代码不止一行的情况。

```
class GameUnit {
    bool IsActive() const;
    //    ...
};

inline bool GameUnit::IsActive() const {
    return m_bACtive;
}
```

应当知道，如果 inline 关键字没有同时出现在函数的声明和定义处，有些编译器会拒绝内联。如果用到的恰好是这样的编译器，最好养成好习惯，在函数的声明和定义的地方都加上关键字 inline。

第二点就是无法保证一个函数真的被内联了。关键字 inline 对编译器来讲仅仅是一个提示，编译器最后才决定究竟是否内联该函数。通常，函数越简单，内联的可能性就越大。一旦函数变得冗长，或者又调用了其他的函数，再或者有复杂的循环，编译器就可能会拒绝为其内联。你也别指望，函数内联在不同的平台上是一致的，因为不同的编译器对内联有不同的规则。

如果一个函数标记为内联，编译器最终拒绝内联，编译器能给出一个警告或者错误就好了。不幸的是，没有那么幸运。大多数编译器如果发现自己不能内联某个函数，它仅仅保持沉默，并忽略 inline 关键字。有两个办法可能查看内联是否成功，一是是否改进了程序的性能，二是查看产生的反汇编代码。这两种办法都不是太方便。但是确认一下一个函数是否被真正内联上是值得的，尤其是对于很关键的代码。

6.3.3 使用内联函数的时机

为什么不总是使用内联函数呢？具有讽刺意味的，它更容易导致游戏的性能下降而不是提升它。好像这还不够糟似的，还有一些关于太随意的函数内联会成为难题的理由。还是从性能开始谈起，毕竟，本章就是讲这个的。

首先，听起来每一个函数都用内联是好像不错的主意。毕竟，这可以去除所有的函数调用开销。第一个问题就是可执行文件的大小会失去控制地猛涨，因为代码所有调用函数的地方都要复制内联函数的代码。先不考虑这会消耗更多的内存，这同样也会导致代码缓存使用得很没有效率，而这会显著降低程序的性能。

滥用内联函数很不好，另外的一个原因是，不得不小心函数定义的位置。前面讲过，一个要内联的函数必须在头文件里定义。这意味着本来写在.cpp 文件里的 include 语句需要移到头文件里去，而这会导致编译时间变长。也许对一个、两个或者很少的几个文件来说，这算不了什么。但是如果一个工程大到有上千的源文件的时候，编译的时间会指数级增长（详细信息可参见第 15 章处理大型项目）。

那最好怎样使用函数内联呢？你编码的时候要避免使用函数内联；然后，当代码快要完成的时候，好好查看程序，看看是否有一些小的程序，但是被调用了很多次，这样的函数就是候选者。这些函数内联之后要立即查看程序性能上是否有提升。

有时候当刚开始写一个函数的时候，就能意识到这个函数应该内联。举例来说，Point 类的某些函数最好内联，因为它们代码量很小，会被反复调用，但是性能指标又很重要。那么立即内联这些函数吧，只要不养成见函数就内联这样的毛病就行。

顽固的、死脑筋的 C 程序员经常问：“内联函数和宏又有什么区别呢？”。毕竟，预处理程序总是期望是宏，不喜欢内联函数，内联函数是由于编译器的慈悲才留下来的。内

联函数的主要优点在于，它仍然和一般的函数一样，做参数数量和类型的检查，而宏就没有一点类型检查。这一点对于想在编译的时候就找出错误，实在是太重要了。内联函数调试也很方便，可以进入函数内部进行单步跟踪；与类配合使用更好，因为它可以成为类的一部分，可以访问类的私有及保护成员，而宏就没有原有类（它所替换掉的类）的一点信息。

有时候内联一个函数可以带来网络性能提高的收获。编译器却固执地不给内联。如果出现这种情况，则不得不回过头来，使用简单的宏（#define）去代换要内联的那个函数的内部代码。这可能不是很好，但是至少宏不会拒绝做这事。

6.4 函数开销更多的方面

函数带来的性能开销问题并不会随着函数调用的结束而完结。传给函数的参数以及函数的返回值都会极大地影响函数的性能。

6.4.1 函数参数

遇到函数参数传递的时候，需要记住最重要的事情就是，永远不要用传值的方式传一个一个比较大的对象。如果这样做的话，会导致对象需要创建一个临时对象的备份，而这个操作的开销是很昂贵的。本章的后面涉及这个话题。除非能保证要传递的对象很小，否则就用传引用的方式传递参数。

下面的函数传递一个矩阵（Matrix）作为函数的参数，用来更新一个节点的包围体。这样的代码能正常工作，但是它还可以再快一点。

```
// Slow version of the function
void SceneNode::UpdateBV(Matrix mat) {
  m_BV.Rotate(mat);
  // ...
}
```

可以重写刚才的代码，使用引用的方式传递参数。从功能上来说和上面的例子是一样的，但是性能会好一点：

```
// Much faster version of the same function
void SceneNode::UpdateBV( const Matrix & mat ) {
  m_BV.Rotate(mat);
  //...
}
```

注意上面的代码里，不仅把矩阵改为引用的方式，而且还把它改成 const 类型的引用。原因很简单，因为函数不允许改变传给它的矩阵参数。把矩阵限定成 const 可以在编译的时候就能得到保证，任何更改它的尝试都会导致编译错误。在第 3 章里已经介绍了。

如果对象很小或者参数就是基本的数据类型，那就没有必要以引用的方式传递它们了。可能取决于具体的平台，但是对于大多数硬件来说，一个引用或者一个指针是 32 位的。除非要传递的对象在构造或者析构的时候开销特别大这种情况，通常没有什么理由需要以引用的方式传递小于 32 位或者更小的参数。

假设所有参数都是基本数据类型的，或者传引用的都是比较大的对象，那传递较多的参数给函数有什么不利结果吗？答案通常是肯定的。但是影响通常很小，这与特定的平台和编译器有关。

一般情况下，编译器会尽可能多地把参数传给 CPU 的寄存器，这样能够要调用的函数被直接访问。有的 CPU 有很多寄存器，所以通常这不是个限制因素。然而，在其他架构的机器上，比如 PC，只有在用完所有的寄存器之前，才有可能用函数的参数填充这些寄存器。剩余的参数就放到栈内。这么做意味着，需要把它们复制到栈的内存某个位置，这决不可能是一个太大的开销（只要对象是以传引用的方式传递的），而且栈是位于数据缓存里，由于它们被频繁使用，所以一般情况下，这不会对程序的性能造成太显著的影响。

然而，最重要的从参数传递过来的不是与性能有关的因素，而是与人有关的因素。计算机才不管我们给函数传递了几个参数呢，它只是把这些参数传递给寄存器，如果寄存器不够用就压入栈。与人有关的因素，就没有那么幸福了。有太多的事情，需要大脑立即做出判断。从经验来看，如果发现自己写了一个函数，而这个函数竟然用了 5~7 个参数，就应该好好考虑，是否有其他更好的办法来解决这个问题。不过，如果几个月后需要改动部分程序或者需要调试这程序，你就会很高兴你改动了函数的参数。办法是把参数封装到一个新的类或者结构体，这样程序就清晰多了。程序甚至还有可能因此运行得稍微快一点。程序结构的清晰和性能不是两个孤立的问题，所以没有理由不做。

6.4.2 返回值

返回一个基本的数据类型，比如整数或者浮点数，这是一件很平常的事。再说一遍，这和平台、编译器，还有函数调用规则有关系。但是典型的情形都是：返回值会复制到寄存器，调用代码再检查这个返回值。

引用和指针，系统是同等对待的。所以只要有可能，需要返回一个曾经在代码里出现过的对象，那返回这个已经存在的对象就是最有效的办法了。只要记住，永远不要返回一个这样的引用或者指针，该指针指向一个在栈里创建的对象。因为，当函数调用结束的时候，该对象就不在那儿了。幸运的是，大多数编译器在编译的时候能够检测到这一点，并报告一个警告。下面的代码就很好的使用了返回一个引用（如果函数使用内联会更好一些）：

```
const Matrix & SceneNode::GetMatrix() const {
  return m_matrix;
}
```

但是，当需要返回一个不存在的对象的时候，怎么办呢？有3个选择：

- 返回对象的备份；
- 在函数外部创建一个对象，以一个非const的引用的参数形式传给函数，在函数里操作这个对象；
- 在被调用的函数内部动态创建对象，并返回该对象的指针。

当需要返回值的时候，第三个办法常常显得很有效率，但是这个办法也有一些不好的方面。主要的问题是，需要动态的创建一个对象，从函数内部来说，这不是坏事情，但是在一个内部性能要求很高的函数里面，这会成为一个很严重的缺陷。此外，如果该函数被频繁执行的话，那就会严重影响堆内存（memory heap），导致内存碎片，从而使性能降低。也有办法避免这种情况，第7章内存分配将详细讲述这个问题。

另外一个主要的问题，第三个办法不会很安全。它在函数内部动态创建对象，并为调用者返回一个指针，接下来，对这个对象的操作（比如释放、保存等）就取决于调用者了。这到后来可能会演化为一个令人头疼的问题。

最清晰而且最简单的办法就是第一个了，但是第二个更有效率。如果这两者同样的快，那返回对象的备份就是了。这种办法，通常可以直接从一个函数返回所有的对象，而不管它们的大小。有时，不得不做其他一些事情，这取决于要调用的类的内部实现。从软件工程的观点来看，这不是太理想。但是有的时候，不得不以小的让步以换取更大的性能收益。

如果在SceneNode的对象里没有一个matrix需要返回它，GetMatrix()函数可能看起来就是下面的样子：

```
void SceneNode::GetMatrix (Matrix & matrix ) const {
  //This object has a rotation object, but not a matrix, so
  //we need to fill the one that gets passed in.
  m_rotation.ToMatrix(matrix);
}
```

如果返回的是matrix的备份，则需要一个额外的备份和一点点的性能损失。

```
//This code is potentially much slower
Matrix SceneNode::GetMatrix () const {
  return m_rotation.GetMatrix();
}
```

第二个版本看起来比较清晰，看起来返回的更像对象的引用，但是它有可能会慢一点，

这取决于编译器究竟是怎样处理的。一个经过很好优化的编译器编译的时候，会在内部把第二个版本的代码转换成第一个版本的那样。所以仍然可以得到最清晰的语法，还有最好的性能。这种优化叫做返回值优化（return value optimization，RVO）。查看编译器的文档，找出它是否支持 RVO。RVO 是目前很普通的优化，所以一般情况下，编译器会支持它的。

有两种类型的返回值优化：命名的返回值优化和非命名的返回值优化。编译器容易实现的是非命名的返回值优化，刚才的例子就是非命名的返回值优化。返回值是临时的（一个未命名的变量），编译器创建它后，直接赋给函数调用结束后要返回的对象。

编译器要支持命名的返回值优化，需要一点技巧。工作原理有点类似非命名的返回值优化，但是需要一个真实命名的变量，不是临时的。这种优化方式要求函数的所有返回路径都要返回同样的值。下面就是编译器可以应用命名的返回值优化的函数的例子：

```
Matrix SceneNode::GetTransformedMatrix () const {
    Matrix mat = m_matrix;
    if (m_pParent != NULL) {
        mat.Concatenate( m_pParent ->GetMatrix() );
    }

    return mat;
}
```

通常，当谈及优化的时候，应当考虑函数到底想达到怎样的效率。需要返回的是 Point3d 对象（也许就是 3 个浮点数而已），而且该函数很少被调用，那返回对象的一个备份就是最好的方案。如果该函数在渲染的每一帧里，都会在一个循环内部被调用很多次，除非知道编译器支持返回值优化，否则就需要传递一个非 const 的对象引用给函数，让函数来给它赋值，虽然这样做有些不便，但是能节省一些 CPU 时间，还是合算的。

6.4.3 空函数

初看起来，探讨一个空函数的性能是一件奇怪的事情，怎么会有人写这样的函数。更奇怪的是，空函数比你想象的要不同寻常得多。

有时候，开发团队有一套模板，可以创建一个新类的草稿。在模板里，声明的函数可能为方便起见，也为了以后少敲一些键盘，它的定义仅为一个空的骨架。还有的时候，一个宏被用来定义为一个类的特定行为，这也有可能包含空函数。

编译器在编译的时候会把空函数去除吗？通常，这和使用什么样的编译器有关系。有时候编译器会去掉空函数，但这不是最普遍的做法。典型的情况是，除非空函数是内联的，编译器没有办法知道函数是空的，从而丢弃它不管。对空函数的调用，和一般函数一样，把参数放到寄存器或者栈里，如果是虚函数还要查虚函数表 vtable，以得到地址，然后执

行函数，最后返回。

明显的建议是不要使用空函数，没有什么好的原因非使用空函数不可。如果创建一个类的骨架用使用的人以后填代码，最好考虑使用纯虚函数。可以建立接口，把函数标记为纯虚的，迫使使用它的程序员实现这些特定的函数。

另外一个常见的错误就是在一个类的继承体系里，我们提供了一些几乎是空的函数，这样的函数的惟一功能就是调用父类的对应函数，不做其他的事情。

虚函数机制已经为我们做好了，如果函数没有被重载，它就调用父类的最近函数实现。这样做也很有效率，在每一次虚函数被调用的时候，能够避免一连串的调用。不仅如此，由于到处添加这样的函数，导致后来改动函数的接口很困难，因为这样的改动将涉及到每一个单独的类的文件，需要改动这些文件的很多地方。

6.5 避免复制

除非没有其他办法了，否则不要复制对象。这听起来很明显，但经常突然出现。一不留神，C++就会背着你复制对象。

不像 C 语言，C++里复制对象的后果可能会大大超出你的预料。从一个已经存在的对象创建一个新的对象，会调用对象的复制构造函数，常常是调用它包含的那个对象的复制构造函数。很显然，复制那样的对象看起来就不是一个很好的办法。

6.5.1 参数

当参数传进传出某个函数时，尽量使用 const 类型的引用。这样的话，语法上和以传值的方式传递一个对象是一样的，但是避免了对象多余复制带来的性能损失。比如要避免声明这样的函数：

```
bool SetActiveCamera ( Camera camera );
```

相反，应该这样声明函数：

```
bool SetActiveCamera ( const Camera & camera );
```

const 关键字对于避免多余的复制不是必需的，它仅是为了说明 camera 对象传递到函数后，不能被函数 SetActiveCamera 所改变。

6.5.2 临时的对象

接下来要做的事情就是查看临时对象了。临时对象是编译器临时创建并使用的，代码里并没有明确的指明。如果不是很了解编译器创建临时对象的规则，则是很难捕捉到的，

而且，由于临时对象的创建经常出现，就有可能显著影响游戏的总体性能。

我们已经看到一处临时对象是怎样创建的：以传值的方式传递对象给一个函数的时候。在代码里这样的复制对象是很明显的，然而还有一种很隐蔽的方式创建临时对象。

一个相当普遍，也很难发现的出现临时对象的类型是由于类型不匹配造成的。看看下面的代码片断：

```
void SetRotation ( const Rotation & rot);
float fDegrees = 90.0f ;
SetRotation ( fDegrees );     //Rotate unit 90 degrees
```

看起来一切正常，也能正常编译，也能像期望的那样正常工作。但是编译器究竟做了什么？毕竟，SetRotation 函数的参数不是 float 类型的。深入地看看 Rotation 类的代码，注意到了它的构造函数：

```
class Rotation {
public:
   Rotation ();
   Rotation (float fHeading, float fPitch = 0.0f, float fRoll = 0.0f );

   //...
};
```

事情变得清晰起来。由于编译器发现传递一个 float 类型的参数给函数，而该函数需要的参数是 Rotation 的对象的引用。编译器能够知道从 float 来创建一个 Rotation 类型的对象是可行的，它就这么做了。这是一个关于隐藏的临时对象的很好例子。

这样的特性对于游戏开发来说是没有什么好处的。也许在一些场合下，这样很方便，可以少敲一些键盘。但是大多数情况下，这样是有坏处的。函数的调用者不想为创建额外的 Rotation 对象付出性能损失。也许代码很简洁，能够看出这样的开销，那程序员可以选择其他的、性能开销相对小一些的函数。

6.5.3 explicit

幸运的是，C++提供了一个特性，可以禁止这样的类型转换。我们可以标记特定的构造函数为 explicit。如果要标记 Rotation 类，那代码看起来可能是这样的：

```
class Rotation {
public:
   Rotation ();
```

```
    explicit Rotation (float fHeading, float fPitch = 0.0f, float fRoll =
0.0f );

    // ...
};
```

这意味着，先前用 float 类型的参数来调用 SetRotation 的代码，在编译的时候会出现一个编译错误。如果确实想用 float 类型的参数，那代码可能是这样的：

```
float fDegrees = 90.0f ;
SetRotation ( Rotation( fDegrees ) );      //Rotate unit 90 degrees
```

最后的代码和第一个版本很像，但是这一次，临时对象是明确可以看到的。任何阅读这段代码的人都可以看出，构造了一个 Rotation 类型的对象，并传递给了 SetRotation 函数。

6.5.4 禁止复制

在游戏开发中，通常会有很多类，这些类是不想被复制的。这和其他类型的软件开发不同，如果复制对象能带来程序的灵活性，一般不会重视程序的性能。对于这些不能被复制的对象来说，我们想要实现，假如有人想复制对象，即使编译器想创建一个临时对象都不行。一个很有用的技术，就是为这样的类提供一个私有的复制构造函数和赋值操作函数声明，但是不去实现这些函数。这样的 C++代码是合法的，可以正常编译，也能正常运行。一旦有人想复制这样的对象，就会导致一个编译（或者链接）的错误。于是，就需要立即修改需要复制这些对象的代码了。

比如，如果有一个几何形体，不想使用复制的办法，因为创建该对象性能开销很大（真是明智的选择），代码可能是这样的：

```
class GeomMesh {
  public:
        //...
private:
        //Private copy constructor and assignment op to avoid copies
        GeomMesh ( const GeomMesh & mesh);
        GeomMesh & operator = ( const GeomMesh & mesh);
};
```

这项技术实在厉害，对于我出先前所说的潜在的性能损失（指避免复制）非常有效。这项技术的完美之处在于，声明这些函数不会带来任何的额外性能开销，这也是一个非常

安全的办法。它很有用，以致于你几乎想在每一个类里都默认地这么用。假使后来需要创建一个类，这个类带有真正的复制构造函数和赋值操作，把它们声明为 public 类型的，再写出它们的定义就可以了。或者注释它们，使用 C++默认的复制构造函数和赋值操作也可以。

6.5.5 允许复制

有时候需要复制对象。这并没有错，尽管知道这样会带来性能的损失。对象越小越简单，使用复制就越安全。一个简单的对象仅包含很少的非基本类型的数据。然而，应该关注可能的不想看到的临时对象问题。如果创建一个通常的复制构造对象或者进行赋值操作，这问题就有可能出现。

有一个不错的替代方案，它仍然可以提供复制对象的能力。那就是手工创建复制函数。通常，想要两个单独的函数：Clone 函数，该函数可以创建一个对象的一个新实例；还有 Copy 函数，该函数能复制内容。

```
class GeomMesh {
public:
          //...
          GeomMesh & Clone() const ;
          Void Copy (const GeomMesh & mesh);
private:
   //Private copy constructor and assignment op to avoid copies
   GeomMesh (const GeomMesh & mesh) ;
   GeomMesh & operator= ( const GeomMesh & mesh);
};

GeomMesh & GeomMesh::Clone() const {
   GeomMesh * pMesh = new GeomMesh();
    PMesh->Copy(*this);
    return *pMesh;
    }

void GeomMesh::Copy(const GeomMesh & mesh) {
    //Do the actual copying here
}

//To use it we just call Clone
GeomMesh * pNewMesh = pMesh->Clone();
```

最后，请牢记，有的时候需要丢弃默认的复制构造函数和赋值操作，编写自己的函数定义这样更好一些。简单的结构适用这种设计，就像非常简单的基本类那样，比如 vectors 或者 points。

6.5.6 运算符重载

运算符重载是 C++的一个特性，也是让人又爱又恨的一个特性。两种观点都有很好的理由，但本节不打算探讨这个。本节的要点是，突出运算符重载可能会带来的风险。

有可能带来潜在性能损失的运算符重载的类型，就是返回一个对象的时候，通常是二元操作符。举个例子吧，运算符+()就是一个这样的运算符。看如下的代码：

```
Vector3d velocity = oldVelocity + frameIncrement ;
Vector3d propulsion = ComputePropulsion();
Vector3d finalVelocity = velocity + propulsion;
```

Vector3d 很明显有 operator +()的函数重载，可以用 3D 的向量相加。代码很清晰，也很直观。不幸的是，代码的最后一行有一个隐藏的临时对象。operator +()的代码看起来大概是这个样子：

```
const Vector3d operator+ (const Vector3d & v1, const Vector3d & v2) {
  return Vector3d ( v1.x + v2.x, v1.y + v2.y , v1.z + v2.z );
}
```

注意到运算符的返回类型不是一个引用或者一个指针，而是一个对象。这意味着，编译器会首先创建一个临时对象，该临时对象将存放着函数的执行的结果，然后再复制给 finalVelocity。

这个例子中，还不是大问题。毕竟，Vector3d 这个类是相当轻量级的，复制起来不会使程序的性能降低太多。这可能是对的，可是这个类被复制的频率比想象的要多。如果这样的代码用在粒子特效系统中用来更新粒子的状态会怎样呢？也许渲染的每一帧都会执行复制上千次，这样事情就完全改观了。此外，operator+()也有可能为其他重量级的类所定义，比如 matrix、rotation，或者游戏对象。

一个好消息是有一个办法可以使我们仍然使用运算符重载，而不会带来额外的性能开销。办法就是把 operator +()替换成 operator +=()，后者会直接操作对象，所以没有额外的临时对象的复制。operator +=()的定义是这样的：

```
Vector3d & Vector3d::operator += (const Vector3d & v) {
  x += v.x ;  y += v.y ; z += v.z ;
```

```
    return *this;
}
```

注意没有任何对象，仅仅是返回一个对象的引用（通过 this 指针）。使用 operator+=() 的代码可能是这样的：

```
Vector3d velocity = oldVelocity + frameIncrement ;
Vector3d propulsion = ComputePropulsion();
Vector3d finalVelocity = velocity ;
finalVelocity += propulsion;
```

或者，写的更好的：

```
Vector3d velocity = oldVelocity + frameIncrement ;
Velocity += ComputePropulsion() ;
```

当使用二元运算符的时候，编译器是有可能优化一些临时对象的；这和返回值优化（RVO）非常相像。在这种情况下，operator +()和 operator +=()的性能就很接近。如果喜欢使用二元运算符，要检查编译器是否有上面的优化，不要等项目快结束了才有惊人的发现。

6.6 构造函数和析构函数

构造函数和析构函数是 C++特性里非常有用的部分。它们关注于对象的初始化或者对象的销毁，这能极大地简化代码，也能降低 BUG 的产生。

然而，像其他自动化的过程相似，有时候它们也会做一些你不愿意的事情，你不期望的事情，或者你没记住的事情。和其他的 C++特性一样，很好的理解它们，会避免大的性能损失。

作为一个快速的回顾，考虑一下像图 6.2 所示的类。

当创建一个 C 的对象的时候，下面是构造函数的执行顺序：

- new 函数的调用为对象分配内存
- A 的构造函数被调用
- B 的构造函数被调用
- C 的构造函数被调用

除了自己的构造函数被调用，如果类 A、B 或者 C 有任何对象，它们的构造函数以及它们父类的构造函数也会被执行。析构函数的执行顺序与此相反：

- C 的析构函数被调用
- B 的析构函数被调用

- A 的析构函数被调用
- 调用 delete 函数以释放内存

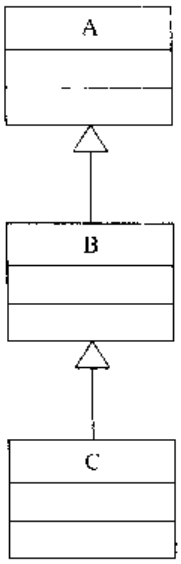


图 6.2 单继承的简单类层次

再说一遍，包含在这些类中的任何对象都是依次被销毁的。这句话的意思是，由于构造函数和析构函数会触发一系列的调用，这可能会给性能带来很大的影响。了解每一个对象可能带来的性能损失，避免在性能非常关键的场合多次创建这样的对象。

然而，C++在创建的时候就充分考虑了性能问题。还有，这个语言的设计目标是：只需为所要使用的那部分支付相应的代价。举个例子来说吧，注意到一个对象中的成员变量在对象被创建的时候，不会自动地初始化为一些默认值；究竟是否给它们赋初值取决于是否需要这么做。有时候，这显得很讨厌，但是这就是为性能所支付的代价。如果没有必要需要构造函数（或者析构函数），编译器就会避免创建一个构造函数（或者析构函数）。

如果很清楚地创建了一个，或者这个对象包含了其他的对象，而这个其他的对象有自己的构造函数或者析构函数，那只会有一个构造函数或者析构函数的调用。同样地，如果某个类属于一个继承链，这个类本身没有自己的构造函数，那在对象的创建过程中构造函数的调用就会被跳过，被调用的是其子类的构造函数。这样的规则也适用于析构函数。

然而，构造函数和析构函数却非常有用。避免在性能很关键的类里不必要的使用它们，但是也不是说在需要的时候压根不用。如果是通过使用一个初始化的函数来实现同样的目的，可能得不到任何性能方面的提升（与使用构造函数或者析构函数相比）。

一个降低构造函数和析构函数调用开销的简单的办法就是使用函数内联。构造函数和析构函数在很多方面和其他函数一样，把它们内联，可以工作得更好。然而请记住，只有那些很简单的函数才能内联。如果它们属于某个继承链，大多数的编译器都需要花费很多时间才能内联上。此外，对于析构的情况来说，任何拥有析构函数，同时又属于某个继承链的类，它的析构函数应该是虚析构函数，这样是为了让内联能够成功实现。

还有一种技术可以降低构造函数的性能开销，那就是使用初始化列表。一个典型的构

造函数是设置成员的值为一些默认值，或者是设置为作为参数传递到函数里的参数的值。一般情况下的函数可能是这么写的：

```
// Inefficient SceneNode's default constructor
SceneNode::SceneNode() {
    m_strName = "Scene node";
    m_position = g_worldOrigin();
    m_rotMatrix.SetUnit();
}
```

这样的代码能够工作，但是构造函数的效率不高。更好的办法是使用初始化列表，这个特性只能在构造函数里使用；它允许为每一个成员对象以特定的参数调用它们的构造函数。通常的做法是：调用默认的构造函数后，然后再改变其状态。使用初始化列表的话，可以节省一次调用。

上面的例子中，变量 m_strName 的类型为 string。在构造函数的第一行代码执行之前，m_strName 的初始值为一个空的 string。接下来，当第一行代码执行后，它的值才被换成一个新值。很明显，string 的默认构造函数的调用纯属浪费时间。同样的情况也存在于 m_position 和 m_rotMatrix。一个更有效的办法就是把上面的函数改写成：

```
// Better SceneNode's default constructor
SceneNode::SceneNode() :
        m_strName( "Scene node" ),
        m_position ( g_worldOrigin ),
        m_rotMatrix( Matrix::Unit )
{}
```

这个例子中，当对象被创建和初始化的时候，所有对象的初始化只出现一次。注意 Matrix 类有一个构造函数可以创建一个单位矩阵（这主意不错）。虽然这不是必需的，有时候这样的工作需要在构造函数里做，不能在初始化列表中做。因为对要初始化的对象来说，不能够作为一个构造函数的调用。这种情况下，只好在初始化列表里不做任何的处理，在构造函数的函数体里初始化就可以了。

上面的例子中，假设 Matrix 类不能通过构造函数设置成一个单位矩阵，那接下来只好在构造函数体里设置了。

```
// SceneNode's default constructor
SceneNode::SceneNode ( void ) :
        m_strName( "Scene node" ),
        m_position ( g_worldOrigin )
{
```

```
        m_rotMatrix.SetUnit() ;
}
```

Matrix 对象没有调用任何特定的构造函数，那就自动调用默认的构造函数。不过默认的构造函数是空的，什么事都不做，没有任何赋值操作。拥有一个默认的构造函数可以避免高昂的性能开销，这对那些被频繁创建而对性能有较高要求的类来说，实在是太有用了。

最后，即使构造函数不能使自己提高多快，它仍然有可能使程序的整体运行快一点。除非彻底没有办法了，否则不要调用构造函数。这说起来很简单，但是对于一个有着很多年编程经验的 C 程序员后来又转过来写 C++程序的人来说，需要有一段适应时间。他需要做的是，直到确实必要的情况下才声明对象。考虑一下下面的代码：

```
void GameAI::UpdateState () {
    AIState nextState;

    if ( IsIdle() ) {
        nextState = GetNextAIState();
        //…
    }
    else{
     ExecteCurrentState();
    }
}
```

除非 AIState 是一个很小的对象，否则函数每一次被调用的时候，它的默认构造函数和析构函数都会被执行。假设该函数在游戏的每一帧都调用该函数，代价就会变得显著起来。使事情变得更糟的是，大多数情况下，游戏 AI 可能不是处于闲置的状态，所以 nextState 这个局部变量大多数情况下，根本就没有用到。不用它的时候，要避免这个开销，只要推后它的声明，在需要的时候再声明就可以了：

```
void GameAI::UpdateState (void) {
    if ( IsIdle() ) {
        AIState nextState = GetNextAIState();
        //…
    }
    else{
        ExecteCurrentState();
    }
}
```

即使 AIState 是一个结构，没有构造函数，最好还是在要使用的时候再声明它。AIState 也许随着时间的流逝发生改变，变成一个大对象了。由于尽可能的推迟对象的声明没有一点坏处，所以不要在函数的开头处就声明。另外，新版本的代码不仅快一点，而且可读性也强一点，因为变量是在将要使用的地方声明的。可以尝试着养成一个好的习惯，所有的局部变量都按照这样的原则去声明，甚至最简单的数据类型，比如 int 或者 float。

6.7 数据缓存与内存对齐（高级话题）

内存对齐在现在的大多数硬件体系结构上都起着重要的作用。CPU 的运算速度越来越快，但是内存存取速度增长的却没有这么快。这意味着，CPU 理论上能达到的速度与由于内存较慢导致其实际能达到的速度存在着差距，而且这个差距在不断地加大。这种模式不大可能很快地改观，所以需要在可预知的未来解决这个问题。

理想的情况，我们希望 CPU 以最大的负荷运行着，一点都没有被内存所拖累而变慢。这就是内存缓充区（缓存）所要做的。缓存是一块很小但是非常快的内存，比系统的主存储器快多了。最近使用的数据和代码就保存在缓存里。缓存能起作用是因为大多数程序不是始终如一地访问数据和代码。大致的规律是，程序 90%的时间是花费在 10%的代码上的。这就是所谓的“局部性原理”（principle of locality）。

有一些硬件结构甚至有着不止一层的缓存，缓存的大小越来越大，但是速度越来越慢。这就是存储体系，这有助于使 CPU 在一般的应用下，以尽可能快的速度运行。

缓存分为数据缓存和代码缓存两种。这里只关注数据缓存。因为可以很好的控制它，也可以从中得到显著的性能收益。

缓存，从定义上来说，它容量很小，不能够保存程序运行时 CPU 需要使用的所有数据。理想的情况是，当 CPU 需要使用某些数据时，它已经在缓存里了。于是可以立即读取数据，CPU 无须等待。另一方面，当需要的数据不在缓存里，就会发生缓存丢失现象，CPU 就不得不等待好几个时钟周期，一直到需要的数据从主存取到为止。CPU 越快，主存越慢，等待的时钟周期就越多，导致程序停止运行，直到数据取回为止。这样的情况要尽可能地避免。

作为一个例子，考虑一下以下的情形：一个主频为 3GB 的 CPU 以满速运行着，当它需要从最快的缓存里取数的时候，只需要一个 CPU 周期。突然，它需要一些数据，而这些数据不在缓存里，所以只好等待从主存取回这些数据。

在当今最快的内存条件下，从主存储器取数并放到缓存里，需要 6 个存储器周期。然而，即使内存总线的频率是 512MHz，CPU 仍然不得不等待 36 个 CPU 周期。如果 CPU 的速度再快一点或者主存的速度再慢一点，那事情会更糟。所以，那些看起来仅仅是访问一个变量的代码可以导致程序的停顿，因为它访问的是一个不在缓存里的变量。其速度将是另外一个看起来几乎一模一样的代码的 1/36，如果这个代码所要访问的数据恰好在缓存里。

幸运的是，可以修改程序，使得尽可能访问的是缓存里的数据，以减少缓存丢失的

现象。

6.7.1 内存访问模式

能够采用的第一个技术就是改变程序数据的访问模式。以游戏里每一帧更新游戏的所有实体为例。通常，想要让每一个实体都有机会更新自己的状态，运行它需要的所有逻辑。

一个很不友好的办法（对数据缓存的连续性不利），那就是一次性遍历所有的实体，找出所有需要更新的实体。接着，也就是第二步，以遍历的方式，逐个更新这些找到的实体。在这种情形下，第一步包括了遍历所有的实体。由于可能有着上千的游戏实体，没有办法让它们都能配合最底层的缓存。因此，每一次访问一个实体的时候，都会导致数据缓存丢失现象发生。这很正常，对此，我们无能为力。然而，让这种办法效率低下的原因是，接下来的又做的第一次、第二次的同样的遍历操作。这意味着，对于每一个实体，都要蒙受一次又一次的缓存丢失。

有一个较好的办法，那就是在一次遍历里面对每一个实体做完它所有的操作。然后再移向下一个实体。虽然仍然有数据缓存的丢失，和上一个办法相比，这个办法的性能要好很多。

然而需要小心这种技术，这种优化技术需要对程序的结构做一些调整。用汇编重写一个小的循环也许是很简单的问题，但这潜在地要求重新考虑算法，改变游戏循环中事件的顺序。也可能为了改善性能，而牺牲一些程序的封装性，但是这只是很多优化技术的代价之一。

最好的办法是，避免最开始的时候做大量的数据传递。不到程序的后期阶段，就不要困扰于内存访问模式。使代码准确地运行，以及代码可读性好，易于维护，这方面的要求要比虽然效率很高但存在错误的代码更为重要。在项目的后期，如果有循环表现得特别慢，怀疑可能有大量的数据缓存丢失。这个时候，为了改善程序的性能，重新审视代码是怎样访问内存的，还是值得的。

6.7.2 对象的大小

讨论缓存的时候，有一件事没有提到，那就是数据缓存的数据的单位。缓存组织成 cache line 的形式，cache line 是缓存的最小单位，可以被缓存进来，缓存出去的最小单位。这在不同结构的体系上表现有所不同，因此可以找出你工作的平台上是多少。对于大多数 PC 来说，一个 cache line 就是 32 字节。

这句话的意思是，如果需要取得一个单个的字节，但是它并不在缓存里，也必须使用 32 字节。这和代码有很大的联系。

首先，对于期望的成百上千的对象来说，它们的处理需要很有效率才行。应该保存它们的大小为 32 字节，或者如果它们不得不很大，那就应该是 32 字节的整数倍。于是，访

问对象的任何部分只需要一个或者两个 cache line 就可以了。

关于这个问题的一个极好的例子就是一个粒子系统里的粒子。我们希望处理数量甚多的粒子，也需要在渲染的每一帧里取得每一个粒子，并更新它们的状态。如果把一个粒子的大小想办法压缩为 32 字节，和 100 字节的对象大小相比，就能获得较好的性能。

6.7.3 内存变量的位置

不仅仅对象的大小重要，对于大多数被访问的对象来说，其保存位置也很重要。数据缓存对 C++对象一无所知，它只知道内存的地址、请求的内存的地址以及 cache line 的地址。意思是说，即使粒子的对象的大小为 100 字节，只要只更新它的前 32 字节，每一个粒子也仅仅使用一个 cache line。所以在那样的情形下，把最常用的成员变量排在前面就非常值得了。

另外，记住有着虚函数的对象有一个 vtable，通常就是对象的最开始的地方的一个指针。如果调用了任何一个虚函数，不得不访问 vtable 指针，这会使对象的前 32 字节放到 cache 里去。这也就是非要把最常用的成员变量放到前面，而不是其他的地方的真正原因。

成员变量在内存中的排列顺序和声明类的时候的顺序一样，所以，要想移动一个成员变量到前面，只要在类声明的时候，把它放到其他成员变量前面就可以了。

6.7.4 内存对齐

如果一个 cache line 是 32 字节，就像到目前为止假设的那样，它在内存中的地址不会刚好是 32 字节。那些 32 字节会对齐到一个 32 字节倍数的地址上。意思是，每一个 cache line 的头部都会映射到一个内存中 32 字节的整数倍的位置上。举个例子来说，如果请求的内容是 0xC00024，cache line 会从 0xC00020 到 0xC0003F 填充数据。

再说一遍，这个特性对怎样放置对象有很大的影响。想象一下下面的灾难性的情形吧，有 1000 个粒子对象，需要在每一帧里更新，每一个粒子对象的大小均为 32 字节。这正好符合 cache line 的要求，好像不错。每一次访问一个粒子，只会有一个缓存丢失(cache miss)，因为它很好的符合了 cache line 的大小要求。

这里有一个大问题。粒子对象在内存里分配好了，所以它们的地址不能保证刚好是从 32 的倍数开始的。从性能的角度来说，这有什么影响呢？

如果一个 32 字节的对象没有对齐在 32 字节倍数的地址上，要访问所有的粒子的所有成员变量就需要两个缓存丢失（cache miss），也会使用两个完整的 cache line。所以即使对象的大小没有变化，正是由于内存对齐的问题，开销上也变成过去的 2 倍。

有的时候，编译器会努力尝试为我们做一些校正，但是在对待具体的某个编译器时，需要注意关于内存校正是否有选项，如果有，就打开这个开关。要不然，动态内存分配可能就以 4 字节为单位做校正了，而这带来的后果会很严重。如果不打算使用我们自己的内

存分配策略的话，关于动态内存分配的内存校正问题，也应该重视起来。这一点在第 7 章会有所讲述。

当对象在栈里分配后，对这些对象的控制力就较小了。它们通常是加到栈的头部，一点也不会考虑它们是否是 32 字节的整数倍。如果想保证对象的对齐，那动态内存分配是一个很好的替代办法。

内存对齐也会影响对象最后的大小。假设有一个对象，大小为 40 字节，如果要为这样的对象创建一个数组，那就会是一场灾难。除非编译器大发慈悲，主动填补了对象的空隙（指凑齐 32 字节的整数倍）。要不然，每一个对象的开始不是 32 字节的整数倍，会导致更多的缓存丢失。在这样的情形下，可能想要把对象的空隙填充，凑到 32 字节的整数倍。于是不管分配的是否是连续的内存，这些对象都位于 32 字节的分界线上。内存对齐是开发高性能程序时需要注意的重要的方面。每一次考查数据缓存优化，都应当验证对象校正是否优化了。

6.8 结 论

在本章的内容里，已经看到了一些关于 C++性能最常见的问题，以及当面对这样的问题时，应当采取怎么样的对策，或者绕过这样的问题。首先看到了 C++所提供的函数的各种类型，每一种类型的性能开销情况，以及在何种场合使用它们。后来又看到通过使用函数内联以最大程度地减少开销。同时也谈到了关于函数性能的其他方面，比如参数传递和返回值。

后来看到，有一些情形，C++会隐性创建临时对象，可能我们都没有意识到，它已经影响性能了。也考查了构造函数和析构函数，以及它们背后可能隐藏的问题。最后，谈到了一个高性能游戏的重要话题：数据缓存的作用，以及在程序中怎样做才能最大程度地发挥数据缓存的作用。

6.9 阅 读 建 议

关于 C++性能的材料实在是太多了，有些书讲述了 C++性能的某些方面，比如临时对象什么情况下会被创建，构造函数的调用顺序，以及不同类型的函数调用的效果。

Meyers,Scott, More Effective C++, Addision-Wesley, 1995.

Meyers,Scott, Effective C++, 2nd ed., Addision-Wesley, 1997.

Bulka,Dov and David Mayhew, Effective C++, Addision-Wesley, 2000.

Pedriana,Paul, High Performance Game Programming in C++, Conference Proceedings, 1998 Game Developers Conference.

Isensee, Pete, C++ Optimization Strategies and Techniques, http://www.tantalon.com/ Petel cppopt/main.htm.

下面是关于返回值的优化的一些材料：

Meyers,Scott, More Effective C++, Addision-Wesley, 1995.

Lippman,Stanley B., and Josee Lajoie, C++ Primer ,3rd ed., Addision-Wesley, 1998.

Lippman,Stanley B., Inside the C++ Object Model, Addision-Wesley, 1996.

下面的这本书是一部关于计算机体系结构的巨著，该书对数据缓存以及内存分级做了很多介绍。

Hennessy,John L., and David A. Patterson, Computer Architecture : A Quantitative Approach, 2nd ed., Morgan Kaufmann,1996.

第7章 内存分配

本章内容概述

- 栈
- 堆
- 静态分配
- 动态分配
- 定制内存管理器
- 内存池
- 万一出现紧急情况（内存耗尽）

大多数C++程序在运行的过程中，需要创建新的对象，需要分配新的内存。应用程序可以选择在栈里分配（栈分配）或者在堆里分配（堆分配）这两种类型。内存分配的每一种类型，在对象是怎么样分配的和它们的生命周期是怎样的等细节方面，都有着自己的特性。

对于一般应用程序的开发者来说，知道这么多就可以了。对于高性能的程序来说，比如游戏或者负荷较高的服务端程序，需要在这方面多加注意。本章主要讲述与堆分配有关的性能问题，还要推荐一些方法和技巧，这些方法和技巧可以用来减轻它们对性能的影响。

通过放弃一些灵活性，会严格限定使用静态分配，以避开所有与静态分配有关的问题。还将看到是怎样在运行期进行有效的完全的动态内存分配的。最后，将看到内存池是怎样以最小的运行期代价帮助动态内存分配，如何避免大多数堆的问题的。

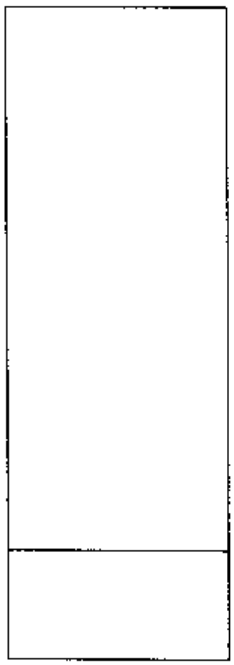
7.1 栈

在程序运行期间，当程序需要创建对象时，如果这些对象仅仅打算在限定的范围内使用，那在栈里创建这些对象就可以了。传递给函数的参数，或者声明在函数内的局部变量，都是在栈里创建的。

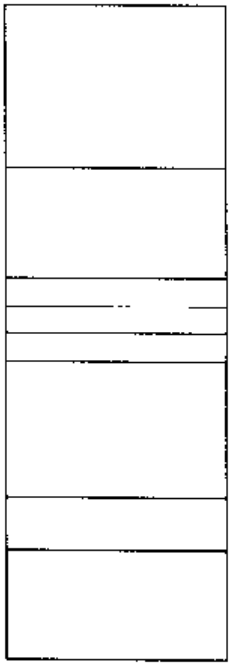
栈是一个好地方。新元素一般追加在栈的底部（或者顶部，如果你的实现是向上增加的），而且它们以进栈时相反的顺序出栈。由于栈有大小限制，企图往栈里加入过多的对象，会导致栈溢出的异常或者其他错误。幸运的是，可以在程序编译的时候，控制栈的大小，在栈使用紧张以致用完空间的时候也可以增大其空间。当前函数的内存地址也是保存在栈里的，所以如果这个地址的内容被破坏了或者被改写了，程序就会停在那里，不知道

怎么返回了。除此之外，栈没有太多的可能性被破坏掉。

图 7.1 展示了栈在分配前的情形及一系列的分配和收回后的情形。



（a）栈在分配前的情形



（b）一系列的分配和收回后的情形

图 7.1 栈

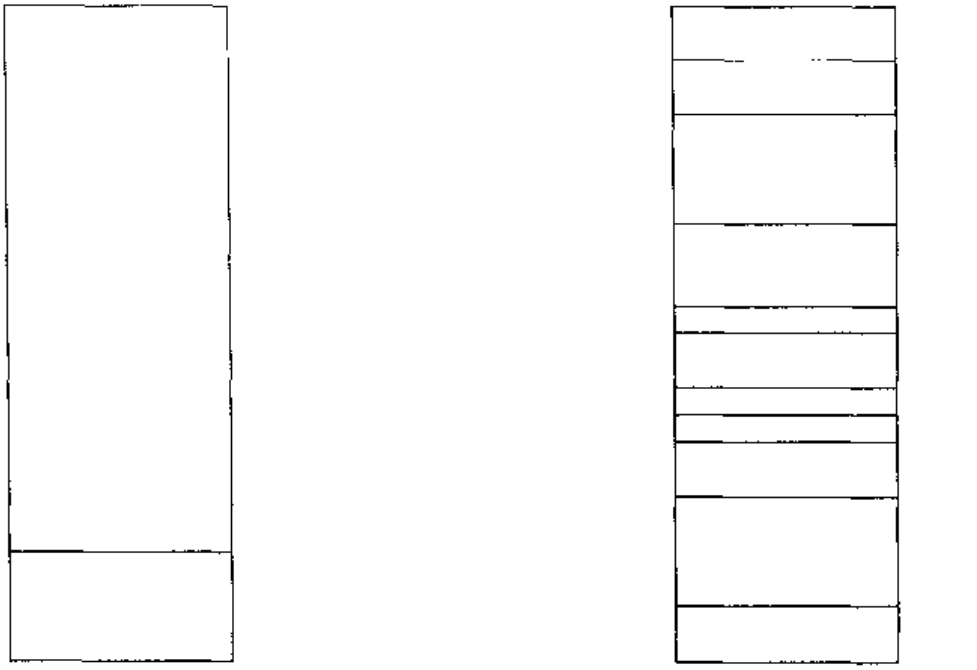
不幸的是，栈有时并不能满足我们的需求。当我们需要创建一个新的对象时，不过这个对象并不是临时的，或者它并没有限定在一定使用范围之内，我们就需要用另外一种办法来分配，这就是堆分配。内存是通过使用 new 或者 malloc 在堆中分配的，分别调用 delete 或者 free 来释放堆中的这些内存。

7.2 堆

栈是一个有次序的空间，而堆就是无秩序的了。这里是内存分配的“狂野西部”。没有任何规则，想怎么干都行。堆是以下麻烦问题的源头：内存泄漏、悬指针、内存碎片等。图 7.2 展示了堆在分配前的情形及在一系列的分配和收回之后的情形。注意，和栈的情况不一样的是，堆的剩余内存空间有很多碎片。

7.2.1 内存分配的性能

有关栈的内存分配性能问题，主要可以归结为从请求内存分配的时刻算起，到内存分配好了需要返回的那个时刻为止，这个过程所经历的时间的长短。这个过程所花费的确切时间和具体平台密切相关，而且会由于具体请求的不同而有很大差异。



（a）堆在分配前的情形　　（b）一系列的分配和收回后的情形

图 7.2　堆

也许你还没有意识到内存分配的性能会成为一个问题，那先考虑一下下面的这个例子吧。游戏在正常运行的时候，帧速率会稳定在 60 帧每秒。这意味着每一帧最多花费 16.7ms，这包括输入处理、AI、物理状态的更新、碰撞检测、网络状态的更新以及图形渲染。假如在有些帧里，要请求分配 500 字节的内存。这样的内存分配要花费 1ms 的时间（每一帧时间的六分之一），分配每一个内存单元需要的平均时间为 0.002ms，这是非常快的。所以很明显，分配内存必须足够快，要不然就不能在游戏里使用。

即使游戏没有要求以一定的帧速率运行，也不能在处理某一帧的时候把大量的时间花费在内存分配上，因为最小的帧速率有时候比平均帧速率更重要，而这将直接关系到用户的感受。

分配内存所花费的时间由于情况的不同差异很大。大多数系统的内存的实现，通常的办法是从表和内存块列表中查找，有时候会有其他一些影响，比如申请内存时，对系统造成的冲击。单个的内存请求常常需要花费好几个百万分之一秒，这是高性能游戏所不能接受的。如果在一台支持虚拟内存的机器上用光了所有的物理内存，那么内存分配所花费的时间将直线上升，达到好几百个百万分之一秒或者更多，因为虚拟内存系统在内存和硬盘之间进行着数据的交换。

同样地，动态内存分配在 CPU 运算速度较快的情况下不大可能获得较好的性能。硬件性能越来越改进了，CPU 的速度越来越快，但是能用的内存容量也变大了，这使得堆的管理更加困难。所以，在可以预计的将来，这依然是一个问题。

7.2.2 内存碎片

另外一个麻烦的问题就是内存碎片。因为堆的性质，内存块以随机的顺序分配和释放的，自然不像栈那样依照着“先进后出”的规则。这样的内存分配模式会导致内存碎片。

在没有任何内存分配之前，堆的原始状态是一块很大的、连续的内存区，不管请求有多大（只要小于实际的内存大小），堆都是同样对待。随着越来越多的内存的被分配和释放，问题来了。由于请求的内存大小是不一样的，被释放的时机也是随机的，最终堆将由一块连续的内存块演变为一块一块的小片断，与此同时，还会有一些比例的没有被使用的内存分散在这些较小的碎片中。

到最后，可能只是需要申请单个的内存，分配都会失败。失败的原因不是由于剩余的空间不够了，而是因为没有一块单独的空间能够满足这样的内存请求。由于这些内存碎片是随机的，所以也没有什么简单的办法可以预测什么时候会发生这样的事情。堆的这个特点能导致难以跟踪的 BUG，而且这样的 BUG，几乎不可能复现。

虚拟地址系统将极大地帮助降低这个问题出现的几率。虚拟地址系统是操作系统或者内存管理库提供的管理内存的另外一种间接手段，这通常依赖于硬件的性能来使得性能的开销几乎为零。工作原理是，使用虚拟的地址而不是内存物理地址。物理地址分成大小相等的小块（比如 4KB），每一块都有一个虚拟的地址，虚拟地址通过一个表格转换成对应的物理地址。这一切对于用户的分配内存的函数来说，都是透明的。关键的一点是，在这样一种机制下，两个原本不着边的物理内存，在虚拟地址系统中可以组成一个连续的内存块（如图 7.3 所示）。于是，前面所说的内存碎片问题也就不存在了。因为可以把很多小的分散的小内存拼成一个大的内存块。

虚拟地址系统带来了一些开销，开销通常不会太大。但是随着内存碎片的增多，查找还有收集未使用的内存会花费更多的时间，这会导致前面所述的显著的性能问题。总的来说，使用虚拟地址系统的好处大大超过它所带来的小的负面影响。此外，像接下来将要看到的那样，我们总是可以控制内存分配，可以绕过所有由于虚拟地址系统造成的代价高昂的内存分配操作。

PC 和 Xbox 都有虚拟地址系统，这是整个操作系统和函数库的一部分。在 PC 平台上，虚拟地址系统走的更远，当内存使用紧张的时候，就把最近很少使用的内存块放到硬盘上，以利于分配新的内存空间的方便。这就是所谓的虚拟内存系统。不用说，对于游戏来讲，这样的情况是不可接受的，因为内存在读写硬盘数据的时候，会导致游戏的速度有明显的下降。

可以在一个原本不支持虚拟内存系统的平台上建立一个这样的内存地址机制。但是这是一项相当费劲的事情，而且要是没有硬件的支持，速度也不会太快。内存的分配和地址映射部分相对来说，要简单一些。这是因为这样的事情：每一个指针以及每一个内存地址都需要转换。这样的转换在程序的代码将是很普遍的事情。由于在指针被使用的时候，没

有办法自动地转换它们，所以要么直接调用 Translate()这样的函数，要么就需要一个能自动转换的指针类。然而这两种办法都不好。当使用第三方的源代码或者函数库，问题会变得更糟糕，因为它使用的不是我们的指针转换机制。

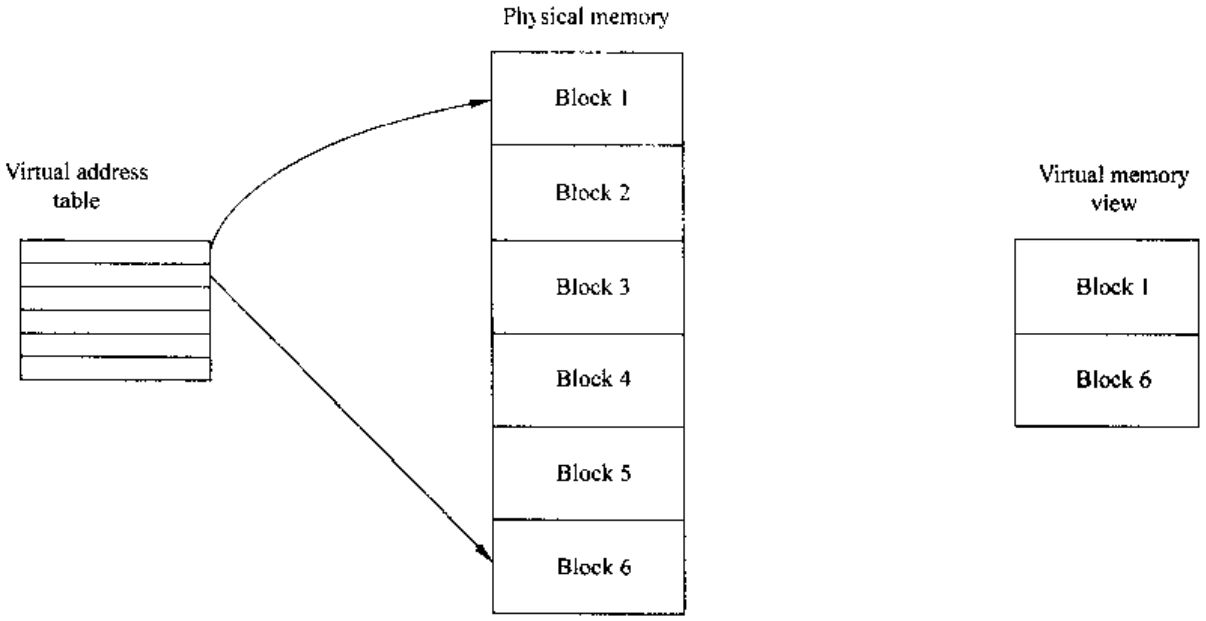


图 7.3　虚拟地址系统把两个原本分离的物理内存块拼成一个连续的虚拟内存块

7.2.3　其他问题

除了上面所说的问题，堆还是其他一些看似与性能无关的问题，然而却是从项目的开始就折磨人的问题的源头。这里提一下，后面会使用一些技术，这些技术是以堆为基础的，需要堆来解决或者提供一些便利来解决问题。

堆的两个最常见的问题是，悬指针（dangling pointers）和内存泄漏。悬指针是一个指针，它用来引用一个有效的特定区域，但是后来由于某种原因，该区域变了，而指针却没有变。当有多个指针指向内存中的同一个位置的时候，假如该内存区域被释放了，却没有更新所有的指针，那就会发生 dangling 指针现象。如果此时试图访问该指针指向的内容，就可能发生以下的事情：

- 如果比较幸运的话，会有一个异常被抛出，程序要么捕获该异常，要么就崩溃。如果指针指向的地方还没有被系统分配，就会抛出异常；如果已经分配给了其他程序，那这个程序就会崩溃。这种情况最好的选择是，让程序捕获这个异常，这相对来说容易跟踪，也容易解决。
- 如果很不幸，这块内存后来被程序重新分配派上其他用途了。如果读这块内存则会返回没有意义的数据。更糟的是，如果是往这块内存写数据的话，就会重写它，还把游戏的部分数据弄丢了。这就是内存超界。它被认为是最难跟踪的 BUG 之一。

这个问题在虚拟地址系统中不是那么明显。由于虚拟地址的范围通常远大于物理内存的范围，所以某个悬指针指向的虚拟地址被重新使用的可能性非常小。大多数情况下，在虚拟内存系统中，访问一个悬指针会立即导致一个异常的抛出。

内存泄漏问题正好相反：有程序分配了一个内存块，但是程序又“忘记”了这个内存块，也就没有控制权返回给系统。这样的后果是明显的。随着程序的运行，内存的消耗会越来越大，内存碎片也越来越多，最终导致内存完全用光，或者如果系统使用了虚拟内存，就会迫使系统把内存倒入硬盘里（这样会导致程序的运行速度变慢）。我们都见过这样的游戏，刚开始运行的时候，一切都正常。在运行半小时后速度就变慢了，最终慢得令人难以接受，程序也几乎停止了，我们不得不从程序中退出，再重新启动游戏，才能正常玩一会儿，再反复以上的过程。这就是典型的内存泄漏的症状。

有时候我们想做的一件事，特别是当要跟踪以上那两个问题的时候，就能得到内存在某一时刻的详细情况。通常想查看在程序运行过程当中，两个指针指向的区域有什么差别，或者仅仅是想看看所有已分配的内存的映像。

有时候，仅仅想得到某个内存块的更为详细的信息：哪一部分代码创建的，什么时间创建的，这块内存有多大等。最后，一个很有用的特性就是可以导出内存的信息，不是显示原始的分配情况，而是一个比较高层的总体情况：有多少内存用于纹理了，有多少用于几何形体上了，或者有多少内存用于寻路算法了。

这种类型的内存情况的报告很重要，尤其对于调整游戏关卡的内存消耗来说更是这样。在游戏终端上比 PC 上更重要，当要跟踪内存相关的 BUG 的时候，这个东西很有用。

7.3 静 态 分 配

对所有的动态分配内存问题来说，最古老的解决办法是避免它们在一起分配。一个程序可以设计成从来不要 new 和 delete（或者 malloc 和 free），仅使用于静态分配的对象或者在栈里创建的对象。下面就是静态分配的一个例子：

```
// Create a fixed number of AI path nodes
#define MAX_PATHNODES 4096
AIPathNode s_PathNodes[MAX_PATHNODES];

// Create a fixed-size buffer for geometry
// BNO
#define GEOMSIZE    (8*1024*1024)
byte * s_GeomBuffer[GEOMSIZE];
```

这个办法有着显著的优点。很显然，动态分配性能不是一个问题，因为它永远不会出现。还有，因为所有的对象都是被编译器静态分配的，而且在游戏运行过程当中也不会改

变，内存碎片和潜在的内存用光的情况也不会出现。

另外一个优点是，静态对象的初始化非常直观，方便跟踪内存以及了解每一种数据类型用了多少内存。在编译的时候就知道任何数组或者缓冲区的大小了，也知道这个大小不会改变。所以只要看一眼代码，就知道内存的分布情况了。上面的例子里，很明显，为几何形体分配了 8MB 的空间，为寻路算法分配了 4096 个节点。

到目前为止，已列出的所有的优点可以解决在本章开头想解决的问题。是不是说静态分配就是要寻找的答案呢？在某些特定的场合下，也许是。但是大多数情况这并不能满足要求。

静态分配内存的主要缺点是会导致内存的浪费。我们不得不提前决定游戏的每一个方面到底需要多少内存，而且这些内存需要一次性地分配好。这意味着对于一个游戏来说，假如这个游戏可能有着复杂的屏幕表现，而且其内容也需要动态改变，那在内存上就有很多浪费。想象一下爆炸效果、粒子特效、敌人、网络消息、寻路的临时信息等，所有这些东西都必须提前创建好。但是如果使用的是动态内存分配，只有当前需要的东西才需要为其分配内存。如果需要的多，那就多分配一些。动态分配内存的方式与静态分配的方式相比，要少用一些内存。

提前决定想分配的对象有多少个，这很重要，但是必须指出的是这还不够。对于每一个单独的类，必须知道需要具体多少个这个类的实例——对象。举例来讲，如果有一个类体系，这个类体系主要是关于对象种类的（比如，敌人、玩家、触发器等）。如果说需要 500 个游戏对象是远远不够的，需要确切地指明需要多少个敌人、多少个触发器等。类的继承体系越复杂，需要的信息就越难描述，也就浪费越多的内存。另外一方面，类的继承体系很复杂可能不是最好的设计。因此，这不是很坏的事，静态内存分配鼓励这样的办法。

静态分配内存的一个突出优点在于，它看起来能减少悬指针出现的机会，因为某个指针指向的内存从来不被释放。然而，它指向的内容却有可能是无效的（比如，导弹爆炸后对象就标记为无效）。此时，指针本身是有效的，但是其内容却是没有意义的数据。这个 BUG 比悬指针的 BUG 还难跟踪。因为使用悬指针可能会立即抛出非法操作的异常。在静态分配内存的情况下，程序仍会继续运行，但由于使用了无效的数据，很有可能在以后的某个时间崩溃。

最后，静态分配内存的一个缺点是，这样的对象需要初始化。这一点不像动态分配的对象，动态分配的对象会在第一次构造的时候初始化，也会在销毁的时候自己关闭。对于静态分配来说，对象只会提前创建，但是没有初始化。这意味着需要额外增加一些代码来初始化所有的对象，以及在不释放对象的前提下，多次关闭掉这个对象。

另外，需要格外小心构造函数里的初始化工作。我们打算为这些对象创建静态数组，你可能会想起，在 C++里，静态初始化是一个“顽固”的问题。一句话，你几乎控制不了对象初始化的顺序。因此，不能依赖于我们的寻路数据准备好，当敌人对象已经初始化或者特效系统已经准备好了，当需要展示特定特效的对象已经创建好了。

事实上，在这种情形之下，最好的选择就是把初始化先放在一边，在构造函数里除了设置一些默认值，标记该对象为未被初始化之外，不要做任何其他的事情。

问题依然悬而未决：什么时候静态分配内存比动态分配内存更可取呢？答案就是取决于具体情况。

一个好的建议是，当没有别的更方便的办法的情况下，就使用静态内存分配。你使用的平台如果使用动态分配内存非常不利于系统的性能，那只使用静态分配内存就是很好的办法了。还有，如果游戏是一个静止的场景，玩家在里面跑来跑去，那么为需要的对象静态地分配内存是有利的。然而，游戏在进行中对交互性可能有更高的要求，这使得静态分配内存更为困难。现在的玩家希望能和他们所处环境中的所有东西交互，比如捡起一个东西，破坏某个东西，移动一个东西，或者创建一个新东西。动态分配内存可以更好地满足这样的情况。

那怎么样才能把这两种方法结合起来使用呢？在游戏的过程中不会改变的对象可以静态的创建，其他一些在游戏的过程中会改变的对象可以动态的创建。把两者结合在一起，看起来可能没有多大吸引力。在某种程度上，这样做会带来最糟糕的问题。对那些动态创建的对象，我们不得不应对它带来的性能问题和碎片问题；对于那些静态分配的对象，我们要保证这些对象有单独的初始化过程和关闭的过程。于是，除非释放的过程是自动的，否则假如忘记了某个对象到底是静态分配的还是动态分配的，而你又以错误的方式释放了它，这将导致更为严重的后果。这样的危险是有可能存在的。

与混合这两种内存分配方式的办法相反，一个最好的办法就是仅仅采用动态内存分配，并遵循本章下面的建议，对于性能敏感的分配使用内存池技术。本章的后面会讲解内存池技术的有关情况。内存池技术可以带来所有的优点和很少的缺点。

7.4 动 态 分 配

当静态分配内存的方式不好用的时候，需要转向更为灵活的动态内存分配。通常，理解它具体是怎么工作的是寻找更为有效解决办法的关键，并希望该解决办法允许我们在运行期使用动态内存分配，而且只要付出很小的性能代价。

7.4.1 调用链

在开始定制内存系统之前，需要理解作为一个内存分配请求的结果，这中间到底发生了哪些事？

1. 每一个东西都是从这样的代码开始创建的：

```
SpecialEffect * pEffect = new SpecialEffect();
```

2. 编译器会在内部替换这样的调用为两个单独的调用：一个是分配正确数量的内存，一个是调用 SpecialEffect 的构造函数。

```
SpecialEffect * pEffect = __new (sizeof(SpecialEffect));
pEffect ->SpecialEffect();
```

3. 全局运算符 new 必须分配要求数量的内存。在大多数 new 运算的标准实现里，全局运算符 new 仅仅是调用了 malloc 函数。

```
void * operator new ( unsigned int nSize) {
    return malloc(nSize);
}
```

调用序列在这里并没有中止，malloc 函数不是一个原子操作。相反，它将调用平台相关的内存分配函数，在堆里为它分配正确数量的内存。通常，这会导致好几个函数调用，还有代价高昂的算法来查找合适的未被使用的内存块。

全局运算符 delete 的处理流程和这相似，但是它调用的是析构函数和 free 函数，而不是构造函数和 malloc 函数。幸运的是，把内存返回给堆与从堆中分配内存相比，通常工作量要少很多，因此就不详细介绍了。

7.4.2 全局运算符 new 和 delete

可以重载全局运算符 new 和 delete 以使之更好地适应目的。但是仍然不能改变内存分配的策略，所以还要接着调用 malloc 和 free 函数。但是可以添加一些额外的逻辑，以跟踪这块刚分配的内存属于谁。稍后，要增加更多的参数，以便更好地控制内存分配。

为了说明内存分配的偏好，先创建一个类 heap。目前，这个类还没有响应请求固定数量内存的函数。它仅仅是一个用来逻辑上为内存分组的途径。开始时所有的 heap 都需要有一个名字。

```
class Heap {
public:
    Heap (const char * name);
    const char * GetName() const;
private:
    char m_name[NAMELENGTH];
};
```

现在准备提供全局运算符 new 和 delete 的第一个版本。

```
void * operator new (size_t size, Heap * pHeap);
void operator delete (void * pMem);
```

此外，全局运算符 new 还需要一个不需要 heap 作为参数的版本。这样的话，所有调用 new 的代码即使没有明确传递一个 heap 进来，程序仍然能正常工作。全局运算符 delete 只有一个版本，该函数要做的工作就是释放掉由任何 new 函数所分配的内存。这意味着不管创建任何形式的 new 函数以及 delete 函数都需要重载它们。

```
void * operator new (size_t size) {
   return operator new(size, HeapFactory::GetDefaultHeap() );
}
```

在具体实现 new 函数之前，先看看它是怎么使用的。要想调用某个特定版本的 new，需要明确为该 new 函数传递一个 heap 引用作为参数。

```
GameEntity * pEntity = new (pGameEntityHeap) GameEntity();
```

必须承认，代码这么写看起来不是很简洁。但是可以改进。先忍受一会吧，保证接下来会改进它的。全局运算符 new 的实现刚开始会很简单。目前来说，想做的事就是保存分配内存的那个堆以及由它分配的内存之间的联系。注意到 delete 运算符仅仅需要一个指针作为参数。所以，从某种程度上来说，需要的另外一个功能就是可以从一个指针得到该指针指向的内存块的信息。

```
struct AllocHeader {
   Heap * pHeap;
   int nSize;
};
```

new 函数和 delete 函数大概像下面的样子：

```
void * operator new ( size_t size, Heap * pHeap) {
     size_t nRequestedBytes = size + sizeof(AllocHeader);
     char * pMem = (char *)malloc(nRequestedBytes);
     AllocHeader * pHeader = (AllocHeader *)pMem;
     pHeader->pHeap = pHeap;
     pHeader->nSize = size;

     pHeap->AddAllocation(size);

     void * pStartMemBlock = pMem + sizeof(AllocHeader);
     return pStartMemBlock;
}

void operator delete (void * pMem) {
```

```
        AllocHeader * pHeader = (AllocHeader*)((char *)pMem - sizeof (Alloc
Header));
        pHeader->pHeap->RemoveAllocation(pHeader->nSize);
        free(pHeader);
    }
```

这两个函数从功能上来说，只是做了最基本的工作。要想做成一个稳定的内存管理器，还需要做很多工作，不过以后可以逐渐地加上。现在这已经是一个很好的开头了。所缺乏的特性是错误检查，检测内存越界的能力以及内存对齐。

即使有那么多限制，现在的这个 new 和 delete 函数已经很有用了。在任何时间都可以遍历所有的 heap，打印出它们的名字，每一个 heap 分配了几块内存，每块内存的大小，内存使用的高峰值以及其他一些有用的信息。我们也有足够的信息可以探测在程序运行当中是否出现内存泄漏。后面能看到它的实现。

到目前为止，我们故意忽略了 new 和 new[]，delete 和 delete[]的紧密关系。new[]和 delete[]的作用是为一个对象数组分配和释放内存。暂时把它们看成不是一个数组，只是调用 new 和 delete。

这个内存管理系统现在刚开始做，也可以用了，但它用起来还是有点麻烦，因为必须为每一个内存分配明确传递一个 heap 作为参数。重载类里的 new 和 delete 运算符可以使这个工作自动完成。最终，在自己的游戏里和工具软件里使用它们。

7.4.3 具体类的 new 和 delete 运算符

到目前为止，我们也忽略了动态内存分配中的函数调用链的另外一个步骤：具体类的 new 和 delete 运算符。当有一个类重载了它们，那对 new 的调用就会调用类的这个 new 函数，而不是全局的那个。这些函数可以保存某些信息，然后分配内容，或者直接调用全局的 new 或者 malloc。

使用具体类的 new 运算符，可以使内存管理器的某些工作自动化。由于通常把一个类的所有对象放到一个特定的堆中，所以可以让类的 new 运算符来处理调用全局的 new 运算符，并传入参数。

```
void * GameObject::operator new (size_t size) {
    return ::operator new(size,s_pHeap);
}
```

以后用 new 来创建一个 GameObject 的对象的时候，就会自动地放到正确的堆里去。这样做就方便多了。那要是每个类都想要这样的功能，我们要做哪些具体工作呢？要做的工作就是为每一个类添加 new 运算符和 delete 运算符，以及一个静态的成员变量。

```
// GameObject.h
class GameObject {
public:
    // All the normal declartions

    static void * operator new(size_t size);
    static void  operator delete(void * p,size_t size);

private:
        static Heap * s_pHeap;
};

// GameObject.cpp
Heap * GameObject::s_pHeap=NULL;

void * GameObject::operator new(size_t size) {
    if(s_pHeap==NULL) {
      s_pHeap=HeapFactory::CreateHeap("Game object");
    }
    return ::operator new(size,s_pHeap);
}

void GameObject::operator delete(void * p,size_t size) {
    ::operator delete(p);
}
```

当多次为类添加同样的功能时，就会意识到需要一个更为简单的办法，以避免敲错代码，程序本身也不够简洁。如果确实需要的话，可以提供两个宏以完成相同的功能，或者使用模板。下面看一下使用宏的版本，上面的类将变成以下的样子：

```
// GameObject.h
class GameObject {
   //Body of the declaration
private:
   DECLARE_HEAP;
};

// GameObject.cpp
DEFINE_HEAP(GameObject, "Game Object");
```

有一个重要的问题，那就是：任何由一个父类派生出的新类都会有着和父类一样的 new

和 delete 运算符，除非它们又重载了 new 和 delete 运算符。在我们的例子中，如果有一个从 GameObject 类继承的新类 GameObjectTrigger，该新类将使用 GameObject 的堆。

现在，我们最终就很容易把新类像钩子一样“钩住”内存管理系统（HOOK，Win32 也有类似的技术），而且为游戏中所有比较重要的类使用这项技术是非常值得的。一个对象可以在程序运行期间做“原始的”内存分配。原始的内存分配是由于从内存中直接分配一定数量的字节数引起的，并没有分配新的对象。如果是这样的话，分配的内存可以被重新指向那个对象所属的类的堆，这在内存使用方面可以保持较好的性能。

```
char * pScratchSpace;
pScratchSpace = new (a_pLocalHeap) char[1024];
```

从这一点上来说，我们已经拥有了一个虽然简单，但是功能上却很完善的内存管理系统的基础了。可以跟踪在游戏运行期间，每一个类或者主要的类类型都使用了多少内存。还可以访问其他一些有用的统计数据，比如内存消耗的峰值。如果再添加一些新的特性，那就可以用于商业游戏的开发了。

7.5 定制内存管理

把前两部分讲到的概念串到一起，就可以创建一个功能全面的内存管理器了。源代码在本书附带的 CD-ROM 上，位置是\Chapter 07. MemoryMgr\folder（Visual Studio 项目文件是 MemoryMgr.dsw），读者可以阅读代码以查看详情。本部分将讲述一些有趣的特性，这些内容是前两部分剩下的，还要添加一些新的有用的特性，并讲述它们是怎样实现的。

7.5.1 错误检查

为了让内存管理器能够在商业软件中得到应用，需要考虑可能出现的错误以及内存误用。前两部分讲的内存管理器并没有准备怎么处理错误。如果不小心传递了一个错误的指针给 delete，内存管理器还会认为这是一个有效的指针，还会删除它。

我们要做的第一件事情，就是确保想要释放的内存是由内存管理器分配的。一般的情况是，对象是使用运算符 new 创建的，使用 delete 删除的。不过有可能，对象使用的内存是由其他的库分配的，或者是代码通过调用 malloc 来分配的。此外，通过检查还可以捕捉指向了错误位置的指针。比如内存被意外破坏的问题，这是指已经分配的内存，其内容后来被意外的改写了。

为了实现以上的设想，需要为 AllocHeader 类添加一个特别的识别标志。

```
struct AllocHeader {
   int nSignature;
```

```
    int nSize;
    Heap * pHeap;
};
```

当然，这里没有什么惟一的数字可以用来作为识别标志，也没有什么数字的组合可以作为识别标志。常常有人以分配的内存数量作为这个识别标志，一般来说，分配了到同一个地址的可能性就非常小。可以增加不止一个整数来惟一标识这块内存，这样会多耗费一些性能，取决于我们的放心程度。但是在大多数场合下，一个整数就足够了。

那这个惟一的整数应该是什么呢？任何不经常出现的数字都可以。比如，使用数字 0 就不是很好，因为 0 在真实的程序中出现的机会太多了。0xFFFFFFFF 也是这样，通常是汇编语言的操作码，或者是虚拟内存的地址。输入随机的十六进制的数字通常就很好，纯粹是为了消遣，有时候取值为 0xDEADCODE。实现的运算符 new 和 delete 看起来是这样的：

```
void * operator new (size_t size, Heap * pHeap) {
    size_t nRequestedBytes = size + sizeof(AllocHeader);
    char * pMem = (char *)malloc (nRequestedBytes);
    AllocHeader * pHeader = (AllocHeader *)pMem;
    pHeader->nSignature = MEMSYSTEM_SIGNATURE;
    pHeader->pHeap = pHeap;
    pHeader->nSize = size;

    pHeap->AddAllocation(size);

    void * pStartMemBlock = pMem + sizeof(AllocHeader);
    return pStartMemBlock;
}

void operator delete (void * pMem) {
    AllocHeader * pHeader = (AllocHeader *)((char *)pMem -
                                sizeof(AllocHeader));
    assert (pHeader->nSignature == MEMSYSTEM_SIGNATURE);
    pHeader->pHeap->RemoveAllocation(pHeader->nSize);
    free(pHeader);
}
```

使用动态分配内存的时候，尤其是处理数组的时候，一个很常见的错误就是写内存越界。为了能够查出这种情况，可以增加一个为记录所分配内存的尾部信息守护变量。只要一个整数就可以了。此外，还可以保存已分配内存的大小，在试图释放这块内存的时候，系统能做双重的检验。

```
void * operator new (size_t size, Heap * pHeap) {
```

```
    size_t nRequestedBytes = size + sizeof(AllocHeader)+sizeof(int);
    char * pMem = (char *)malloc (nRequestedBytes);
    AllocHeader * pHeader = (AllocHeader *)pMem;
    pHeader->nSignature = MEMSYSTEM_SIGNATURE;
    pHeader->pHeap = pHeap;
    pHeader->nSize = size;

    void * pStartMemBlock = pMem + sizeof(AllocHeader);
    int * pEndMarker = (int *) (pStartMemBlock + size);
    *pEndMarker = MEMSYSTEM_ENDMARKER;

    pHeap->AddAllocation(size);
    return pStartMemBlock;
}

void operator delete (void * pMemBlock) {
    AllocHeader * pHeader = (AllocHeader *)((char *)pMemBlock -
                            sizeof(AllocHeader));
    assert (pHeader->nSignature == MEMSYSTEM_SIGNATURE);

    int * pEndMarker = (int *) (pMemBlock + size);
    assert (*pEndMarker == MEMSYSTEM_ENDMARKER);

    pHeader->pHeap->RemoveAllocation(pHeader->nSize);
    free(pHeader);
}
```

最后，作为另外一个安全守护的办法，一个好的策略就是填充要释放的内存为一些明显与众不同的字节位。于是，当意外地改写了这块内存的任何一个部分，都能立即看到，这是由于要释放内存引起的。还有附带的好处，如果访问了一个已经释放的内存区域话，填充的字节位的内容恰好也是程序要终止的操作码，那么程序就会自动终止。

到现在为止，我们增加了额外的开销：为扩展分配头信息和尾信息。这和分配内存的操作和释放内存的操作一样。由于最初的目的是创建自己的内存管理器，并获得更好的性能，然而这看起来像背道而驰。幸好这里所做的一切，都是仅发生在调试版本。后面将会看到，新的开销大多数在发售版本都不会出现。

7.5.2 遍历堆

有时候，为了检查内存的连续性，能够遍历一个堆的所有内存分配区域，这是一个必要的需求。这样做可以收集更多的信息，还能够消除内存碎片。问题是，到目前为止，还

不能遍历一个堆。还需要为 allocation header 附加一些新的信息：

```
struct AllocHeader {
   int nSignature;
   int nSize;
   Heap * pHeap;
   AllocHeader * pNext;
AllocHeader * pPrev;
};
```

我们为这个结构加了两个字段：pNext 和 pPrev，它们分别指向下一个分配区和前一个分配区。在一些场合下，这比使用 STL 的 list 效果要好一些。但是这是一个很底层的系统，最好还是用这种方法实现它，以避免 STL 可能带来的多余的内存消耗。

在每一次内存分配和内存释放之后，运算符 new 和 delete 会小心调整 list 的指针，由于要维护一个双向链表，而这个双向链表的开销很小，可以忽略掉。现在就可以遍历整个堆了，主要从第一个分配区开始，一个一个的按着顺序访问它的下一个分配区就可以了，一直到最后。

7.5.3 内存书签和内存泄漏

拥有自己定制的内存管理器让人愉悦的其中一点是，你可以做任何你想做的、你需要做的事。一般都需要跟踪内存泄漏，所以还是先修改代码来支持这个特性吧。

找出内存泄漏从原理上说很简单。在某个时刻，给内存的使用情况做个标记（内存书签），过一会，再做一个标记（内存书签），同时再列出第二次分配的内存（不是第一次分配的）。不用吃惊，这个实现就是这么简单。

我们要做的也就是加一个保存分配次数的计数器而已。每一次分配内存的时候，就递增这个变量，已用内存加上刚刚分配的内存的大小就可以了。在这个例子里，假定分配的内存永远不会超过 2^{32} 次。如果有可能超过，就需要用一个 64 位的数字来记录分配内存的次数，或者再另外设计新的办法。不管哪种情况，实现起来都很简单。allocation header 现在大概是这个样子：

```
struct AllocHeader {
   int nSignature;
   int nAllocNum;
   int nSize;
   Heap * pHeap;
   AllocHeader * pNext;
AllocHeader * pPrev;
};
```

运算符 new 和过去差不多，改动之处就是给 nAllocNum 赋值。另外，增加了一个函数，GetMemoryBookmark。它的惟一任务就是返回当前已分配的内存。

```
int GetMemoryBookmark () {
    return g_nNextAllocNum;
}
```

函数最后做的一件事是报告内存泄漏。该函数需要两个参数，分别记录内存地址的开始和结束，函数能报告出这段内存区间内，依然有效的已分配内存块。该函数的实现很简单，只要遍历堆里的所有的已经分配的内存块，看看其地址是否在指定的内存起始地址和结束地址之间就可以了。是的，这样遍历所有的堆，查看每一块已分配的内存会很慢，但是没有关系，我们仅仅是在调试状态下才使用这样的函数，不需要关注程序执行的有多快。这个查看内存泄漏的伪码可以表示如下：

```
void ReportMemoryLeaks ( int nBookmark1, int nBookmark2 ) {
   for ( each heap ) {
         for ( each allocation ) {
               if ( pAllocation->nAllocNum >= nBookmark1 &&
                    pAllocation->nAllocNume nBookmark2 ) {
                    //  Print info about pAllocation
                    //  Print its alloc number, heap ,size ...
               }
         }
   }
}
```

7.5.4 有层次性的堆

前面讨论堆的时候留了一个方面没有说，那就是堆的层次性。这个东西很不错，对于规模比较大的程序的开发会非常有用。

堆的使用会激增。一开始是所有的图像处理使用内存，后来演化成 15 个不同的堆处理不同的方面：一个用于顶点（vertex）信息，一个用于索引，一个用于阴影（shader），一个用于纹理、材质、mesh 等。也许还没有意识到，游戏已经使用上百个堆了。此时，要想找到有关的信息，这个过程就会变得很慢、很烦人。此外，有时仅仅想要一个大点的图片，也许美术总监就会问“我们的图形数据总共有多大的内存可供使用？”

这时就需要引入堆的层次性了。像到目前为止看到的那样，它们仅仅是有规则的堆而已，但是它们组织成树的形式。每一堆都有一个父亲，可能有一些孩子。惟一不同的是，

每一个堆都保存有自己的统计信息以及它所有孩子的统计信息。

看一个例子，如表 7.1 所示为没有使用层次结构的堆内存的报告情况。

表 7.1　没有使用层次结构的堆内存情况

堆	内　存	峰　值	INST
顶点	15346	16782	1895
Index lists	958	1022	1895
纹理	22029	22029	230
材质	203	203	321

使用了层次结构的堆内存情况如表 7.2 所示。

表 7.2　使用层次结构的堆内存情况

堆	本　地			全　部		
	内　存	峰　值	Inst	内　存	峰　值	Inst
渲染	0	0	0	38536	40036	4341
几何形体	0	0	0	16304	17804	3790
顶点	15346	16782	1895	15346	16782	1895
Index lists	958	1022	1895	958	1022	1895
材质	0	0	0	22232	22232	551
材质对象	203	203	321	203	203	321
纹理	22029	22029	230	22029	22029	230

从程序实现的观点来看，惟一的不同在于，需要说明要创建的堆在层次体系的哪个位置上，以及要跟踪它的父亲和孩子。这一点也不复杂，就是一些指针操作而已。具体情况可参阅本书附带的光盘（\Chapter 07. MemoryMgr\MemoryMgr.dsw）。

7.5.5　其他类型的内存分配

很不幸，重载全局的操作符 new 和 delete 并不是结束。需要小心使用所有的 new 和 delete 调用，不管是全局的那个还是特定类重载的那个。除此之外，还有其他类型的内存分配。

直接调用 malloc 不会妨碍使用内存管理器，调用平台相关的内存分配函数（比如 Win32 下调用 VirtualAlloc 或者 HeapAlloc）也不会影响。在这样的情况下，除了尝试着手工修改之外，没有别的办法。

一个可能性是为 malloc 函数创建一个可定制的版本。新的 malloc 函数需要堆指针，将来直接调用新的 malloc 函数就可以了。只要可以得到源代码，只要不在太多的场合调用它，这个定制的版本的 malloc 函数就可以工作得很好。

另外一个替代办法，特别是在无法取得源代码的时候，那只好手工跟踪内存的分配了。在要调用这个无法控制其内存分配的函数之前，先取得整体内存的状态。然后调用该函数，再找出分配了多少内存，接下来再分配同样数量的内存给一个特定的堆。

```
int nMem1 = GetFreePhysicalMemory();//Platform-specific call
//  Made-up function that will use platform-specific
//  memory allocation functions
DirectXAllocateBuffers();
int nMem2 = GetFreePhysicalMemory ();
PDirectXHeap->AddAllocation( nMem2- nMem1 );
```

只有在调用不多的情况下，只有在什么时候分配内存，什么时候释放内存都很明确的情况下，上面的代码才能起作用。如果调用的函数要缓存一些已分配的内存，那上面的代码就无法保证可靠性了。

有一些 API 设计的就很好，可以被内存分配函数“钩住”。这些 API 允许你提供一个对象（或者一系列函数指针，如果不习惯 C++的形式）。每一次 API 需要为堆的内存分配做任何形式的排序的时候，这样的对象都会被调用。在这样的情况下，可以创建一个对象，调用运算符 new 和 delete，并使用某一个特定的堆来分配内存。所有的用 API 来分配的内存都可以在堆里得到跟踪。

7.5.6 内存系统的开销

为定制的内存管理器添加了很多有用的特性，这样的特效有助于监视内存的使用以及性能方面的问题。但是，还有一个问题，为了保存跟踪内存的信息，也为每一次的内存分配增加了几个字节数。几个字节看起来没有什么大不了的，其实不然。内存分配的结构大致是这样的：

```
struct AllocHeader {
   int                nSignature;
   int                nAllocNum;
   int                nSize;
   Heap *             pHeap;
   AllocHeader * pNext;
   AllocHeader * pPrev;
};
```

假设每一个整型数字或者指针类型的变量在内存中占用 4 个字节，那这个结构使用了 24 个字节。后边会看到，在某些特定的平台上，想把这个数字凑成 32 字节，因为这样可

以为要返回的内存分配改善内存对齐的性能。另外，还要为表示每一次分配的内存块的结束标志附加4个字节，所以加起来达到36个字节数。

如果动态分配内存的次数很少，而且分配的都是较大的内存块，那绝对能够接受这样的开销。但是如果都是这种情况，那本章就完全没有讲述的必要了，因为这种情况下，动态分配内存的性能以及内存碎片现象就不是问题了。正如本章开头所述，C++鼓励较小的但是很频繁的堆内存分配，所以开销问题立刻变得重要起来。这个问题在有着内存数量限制的游戏终端（游戏机）上更为严重。

一个设计成最大使用64MB内存的C++游戏，可以很容易地分配50 000或者更多个堆内存。如果每一次分配使用36字节，使用内存管理系统会额外有1.7MB的开销。如果工作的平台限制有内存限制，并且最大内存为64MB，那这个办法就有点接受不了了。

在调试状态下，事情会变得更为糟糕。如果实现了全局的运算符new，且调用了malloc函数的话，那么malloc会多出32字节的开销（取决于特定的malloc实现）。这样总的开销会达到3.2MB。

幸运的是，可以用malloc所做的那样，不要这样的信息。在调试状态下记录必要的信息是非常有用的，但是在发行版（release版）就没有必要了，发行版是指直接可以给销售商的可执行程序。所以在发行版的情况下，就没有必要记录任何信息了——内存分配的开始和结束标志等。malloc函数也不会保存任何额外信息，所以在发行版的情况下不存在开销问题。

```
void * operator new (size_t size, Heap * pHeap ) {
#ifdef _DEBUG
  //     Same implementation as before
#else
  return malloc(size);
#endif
}

void operator delete (void * pMemBlock, size_t size) {
#ifdef _DEBUG
  //    Same implementation as before
#else
  free ( pMemBlock);
#endif
}
```

惟一的缺陷是，所有的错误检查在release版本里都没有了。意思是说，内存系统不能立即检测出内存越界，或者传递了一个无效的指针要删除，这会导致出现各种各样的不好的情况。这是否能够被受取决于具体情况，开发的游戏的具体类型，或者使用的平台（可参阅第16章来了解更为详细的信息。在这一章里，还可以了解如何避免游戏在运行过程中

崩溃，以及游戏的release版本有错误发生时，应该怎么办等方面的问题）。

7.6 内 存 池

关于使用内存管理，已经介绍了内存是怎样逐渐用光的，如何跟踪内存泄漏，如何避免不必要的内存的开销。不过还有另外一个问题没有探讨，那就是性能问题。一开始开发内存管理器的时候，一个动机就是要获得比直接调用new或者delete更好的性能。到目前为止，用一个new、delete替换掉了malloc、free（不过，malloc和free是内存管理库的底层使用的分配和释放内存的方法），并没有对性能有任何改进。对于大多数内存分配问题，要想改进其性能，一个常见的办法就是使用内存池。

回忆一下堆内存分配的默认实现，其代价最为高昂的地方就是查找出要返回的内存块。特别是当内存碎片特别多的时候，查找算法不得不遍历每一个小块，直到找到合适的可以分配的内存块为止。

从概念上来讲，内存池是一块预先分配好的内存，其内存大小一定，可以在程序运行的时候用来为对象分配一定数量的内存。一旦这些对象被程序释放，它们使用的内存并不归还给堆，而是归还给了内存池。这个办法有很多优势。

- 没有性能上的损失：一旦有程序从内存池申请了内存分配的请求，内存池会从预先分配好的内存中返回第一个没有使用的块。没有调用malloc，也没有使用查找。
- 没有内存碎片现象：内存块仅分配一次，从不释放，所以在程序运行的过程中，堆内存不会发生碎片现象。
- 一次性分配很多内存：可以以任意的方式预先分配内存块。典型的例子就是，可以预先分配一块很大的内存，只是在需要的时候（申请内存分配时）返回其较小的分区。这样做的好处是减少了堆内存分配的数量，也为要返回的数据提供了空间上的连贯性（由于改进了数据缓存的开销，从而也获得了更高的性能）。

使用内存池惟一的缺点在于，由于内存池开始分配了很多内存，但是通常情况下又用不完，所以会有一些浪费。不过，只要内存池的大小是合理的，也仅仅用于动态的对象，那这多出来的一点内存和所获的性能相比，还是物有所值的。

使用内存池，还有一个附带的好处。可以一次性地释放掉内存池中所有的对象，而无须调用它们的析构函数。很明显，这样做时要小心一点，需要保证内存池之外确实没有人引用这些对象。一旦明确了这一点，内存池就可以擦去其中的所有数据。这样做的原因很简单，就是效率：一次性的擦去所有的数据要比一个一个地析构内存池中的每一个对象快得多。游戏中可以用到这个技术的地方有两个方面，一是退出某关卡的时候，清掉大块的内存；二是用于销毁大量相关的较小的对象，如果这些对象需要立即消失（如粒子特效、寻路节点等）。

7.6.1 内存池的实现

首先，为内存池创建一个类，然后内存系统再“钩住”它。这个类需要知道的第一件事就是它要分配的对象到底有多大。由于这一般不会有变化，可以以参数的形式传递给构造函数。另外，需要为内存池实现分配和释放内存这两个基本的函数。这是第一步，内存池类的最简单版本。

```
class MemoryPool {
public:
   MemoryPool ( size_t nObjectSize );
   ~ MemoryPool ();

   void * Alloc ( size_t nSize );
   void Free (void * p, size_t nSize );
};
```

我们需要提出内存管理的一种模式，来管理这些很多相似大小的内存块。调用 Alloc 成员函数后返回一个内存块，当调用成员函数 Free 后再放回原处，继续受我们控制。我们可以预分配所有这些内存块，并用一个链表来保存这些内存块的指针。当调用 Alloc 函数的时候，就返回链表的第一个内存块的地址。当调用 Free 函数时，就把这块内存归还给链表。从概念上来说，这很简单。这能够避免在运行期动态分配堆内存；当然与此同时，这个方法也存在一些问题。最主要的问题就是增大了每一次内存分配的开支。现在每一次内存分配都必须找到链表里的没有被使用的内存块，同时也意味着所有的事先分配好的内存块都是单个的，不是一个大的内存块。因此，这样做也导致操作系统需要多次分配这些小的内存块。

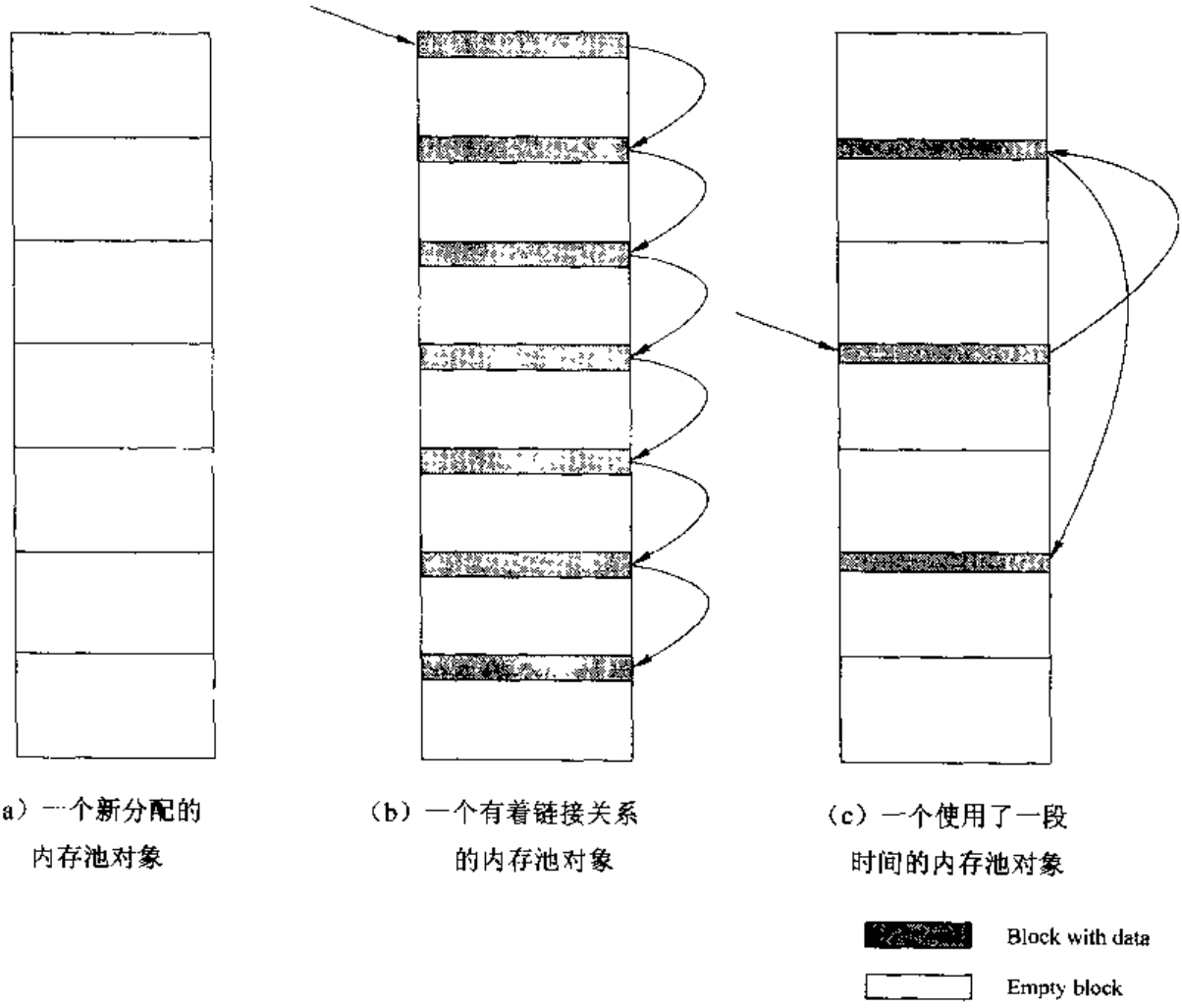
还有一个好一点的办法可以解决所有这些问题，但是需要多费点劲：直接处理内存，映射内存地址到特定的数据以及其他不可见的东西。虽然麻烦，但是这样做绝对是物有所值。最后，并没有多余的开销，所有的内存分配都发生在一块大的连续的内存块里，这正是所期望的。另外，所有的复杂性都被隐藏在内存池这个类的内部。因此要想使用它，没有必要非得了解它是怎么实现的。还有一个事实，如果处理的比较好的话，使用者甚至根本没有必要知道这是用内存池这个类来实现的。

先分配一个大的内存块。可以认为这个大的内存块是由一系列小的连续的内存片组成的。每一个小的内存片就是分配的单位，请求内存池分配内存的时候，分配的就是这样大小的内存块，如图 7.4（a）所示。

一开始，在没有任何内存分配之前，内存块所有的内存都是可用的，因此所有的内存片都被标记为未使用。为了能够管理这些内存片，需要一个链表。因为内存块已经分配好

了，但是还没有分配给任何程序，这些内存全是没有被使用的内存，应该好好利用。我们不会为已使用的内存和未使用的内存各建一个链表，这样太浪费内存了。只建一个链表就可以了，只是每一个内存片都有 next 和 previous 两个指针，可以把所有的内存片用一个双向链表组织起来，如图 7.4（b）所示。

剩下的就简单了。需要向内存池请求内存分配的时候，首先在这个空闲内存片链表里找到第一个元素，整理好链表（修改指针而已，把刚刚找到的这个元素从链表中去掉），返回这个内存片的指针就可以了。一旦内存要释放了，就把这块内存加到空闲内存片链表的头部。经过几次分配之后，内存片的顺序就乱了，不过没有关系。不会导致任何内存碎片问题，而且可以以相当快的速度分配和释放内存，如图 7.4（c）所示。



（a）一个新分配的内存池对象　（b）一个有着链接关系的内存池对象　（c）一个使用了一段时间的内存池对象

图 7.4　内存池的实现

了解了内存池这个类是怎样工作的这个问题之后，映入脑海的第一个问题就是，如果需要更多的内存，怎么办？答案很简单。分配一个和第一个内存块相似的内存块，然后把它们链接到一起，把新分配的内存块加到空闲内存片链表就可以了。

接下来的问题就是，这样的内存块应该要多大才合适呢？这是一个很难解答的问题。

这完全取决于执行的内存分配的类型，以及使用内存池的应用程序的具体要求了。一方面，如果分配太大的内存块，也许以后永远用不完，我们也不想浪费这么多空间。但是另外一方面，也不想由于分配的内存块太小导致分配操作过于频繁。

最好的实现就是使用默认值。举个例子来说，每一个内存池都创建512个小的内存片。这里实际上没有必要非得是2的多少次幂，但是这样程序运行会好一点。不过也没有关系，可以改成想要的数字。一旦内存管理器底层使用了内存池，就可以报告出浪费了多少内存，自然就可以处理它了。可能也想确认那些已分配的内存块的大小是内存页的整数倍，也希望分配的速度快一点。在Win32平台上，每一个块是4KB的整数倍，而且通过调用malloc函数就可以直接分配内存。

一旦开始内存分配，内存对齐的问题就来了。在一些平台上，为得到最好的性能，我们设计的结构，其大小应该对齐到某个特定的内存边界值。比如说，在Win32平台上，那最好校正到32字节的分界线上。假如决定创建一个内存池，要为一个大小为78字节的对象分配内存，那创建内存片的大小接近32字节的整数倍是有意义的。这个例子里，就是96字节。于是，所有内存池分配出去的内存均被校正为32字节。至少应该考虑是否为堆这个类添加一个选项，来强制特定的校正。内存管理器也应该把额外浪费掉的内存报告出来，所以时刻要记得那些分配的内存都到哪里去了。

7.6.2 钩住它

到目前为止，内存池这个类实现了很多功能，但是用处并不大。它仅仅是分配原始内存，所以还不能通过它为一个对象分配内存。而且，使用起来也太麻烦，因为需要清楚地表示出先取到这个内存池，然后调用Alloc函数，并传入需要的内存的具体大小。

改进一下这个内存池类。可以重载类的运算符new和delete。为确保特定类的所有对象使用内存池为它们分配内存，需要为类添加一个静态的成员变量，为了能够使用它分配内存，我们还要再重载运算符new和delete。

```
// MyClass.h
class MyClass {
public:
   // All the normal declarations ...

   static void * operator new ( size_t size );
   static void operator delete(void * p, size_t size );

private:
   static MemoryPool * s_pPool;
};
```

```
// MyClass.cpp
MemoryPool * s_MyClass::pPool;

void * MyClass::operator new ( size_t size ) {
   if ( s_pPool = = NULL) {
        s_pPool = new MemoryPool( sizeof( MyClass) );
   }
   return s_pPool->Alloc( size );
}

void MyClass :: operator delete (void * p, size_t size ) {
   s_pPool->Free( p, size );
}
```

每一次由 MyClass 创建对象的时候，都会使用内存池的内存。这不错，但是对于每一个想使用内存池的类，不得不敲太多键盘，还是有点啰嗦。如果使用宏，那会省很多事，代码也会整洁很多。改进后的代码如下所示：

```
// MyClass.h
class MyClass {
public:
   // All the normal declarations ...

MEMPOOL_DECLARE( MyClass )
};

//  MyClass.cpp
MEMPOOL_IMPLEMENT( MyClass )
```

代码清晰多了。只需要加上那两行，就可以让任何类使用内存池。

最后，要想把内存池整合到内存管理器中，每一个内存池都需要一个堆，还要通过堆注册每一个分配器。现在所有的分配器，不管有没有使用内存池，都会在内存池中显现出来。源代码参见随书光盘（\Chapter 07.MemoryMgr\MemoryMgr.dsw），代码也包含前面所说的宏定义以及最终版本整合了内存池的内存管理器。

7.6.3 Generic Pools

如果有一系列的类构成一个继承体系，而且每一个类都的大小都不太一样，那会怎么样呢？任何一个从 MyClass 继承而来的类都使用同一个内存池，由于它们的大小不尽相同，

所以当试图实例化一个对象的时候，都会抛出断言。很显然，不能仅仅从它继承就以为万事大吉了，别指望它能正常工作。

幸运的是，大多数需要使用内存池的对象都是很简单的结构，没有其他的类（或者结构）继承它。这种情况下，内存池可以很好的工作。这样的对象一般是句柄、消息或者一个限定了结构的节点（比如 BSP 或者 QuadTree）。

另一方面，也许已经有了游戏对象的继承体系，它们的大小也是不同的。我们希望能使用内存池，因为它们在程序运行的过程中频繁地比较内存会影响程序的性能。在这种情况下，可以创建一个一般的内存池（generic pool）分配系统。不是每一个类都有一个自己的内存池，内存管理器有固定数量的内存池——大约 5~6 个。在这以前，使用内存池都是分派特定大小的内存，这里将使用这些内存池来分配那样大小的内存或者更小一点的内存。当内存池收到一个内存分配的请求时，内存管理器会在堆里查找能满足请求的最小的对象，然后就在那里分配。

很显然，这种机制会导致更大程度的内存浪费，不过如果有很多不同的对象需要在运行期创建的话，这么做还是值得的。拥有好的性能，尽可能的少浪费内存，这才是最重要的。另外，通常为大于 4KB 的对象在堆里分配内存是不值得的，因为这样的分配在游戏过程中并不频繁。操作系统通常对分配大的内存块很在行，效率也比较高。

注意 generic pools 可以一个接着一个，挨着通常的内存池或者更一般的没有使用内存池的内存块。以堆为基准，可以自由选择内存分配的类型。因此，可以选择适应具体工作内容最好的工具。

7.7 万一出现紧急情况（内存耗尽）

你已经决定在游戏里使用动态内存分配了。你实现了一个内存管理器、一个内存池系统，还有你已经把这加到你的游戏引擎里了。你得到了使用动态内存分配的所有灵活性，而且没有什么缺点。不幸的是，有一件事你必须要时刻注意，那就是内存耗尽。

你可以不管它，但是最好还是严肃对待这个问题。除非你特别留意什么时候，怎样分配每一个单个的动态分配。玩家可能出现这样的情形：那就是所有的动态内存分配在同一时刻发生了，这时系统也许会用完所有的内存。这可不是好玩的事情。有两个办法可以对付这样的问题：避免发生这样的问题，或者一旦发生了，怎么处理它。

可能最自然的办法是避免内存用完，这才是上策。可以实现这一点，在分配内存的时候记下已经使用了多少，然后和系统能提供的内存相比较，但是这样会有一点性能上的损失。比如说，如果系统里内存不多了，那将要产生的粒子系统里的粒子的数量就应该适当减少一点。如果内存还是不断减少，而且到了警戒点，那就应该不要再生成这个粒子系统了。虽然游戏看起来不那么眩目了，但是可以避免游戏崩溃。另外，还要算上临时使用的内存，这样的内存也许就在几帧后又降下来了，可能这时，系统已有的内存可以满足产生所有的粒子的需要的内存了。可以在其他系统里使用类似的机制，比如声效系统、AI 计算，

或者其他需要动态分配内存的系统。使用这样的机制，可以避免游戏崩溃。

另外的替代方案可以让系统用光内存，然后再处理它。这个更复杂了，首先，在每一个分配内存的地方都检测，是很奢侈的事情。直到一个对 new 函数新的调用如果失败了，就需要处理了。

一个通常的策略是，在程序开始的时候就分配一些内存但是不使用，比如 50KB 或 100KB。 一旦用完了内存，要做的第一件事就是先释放掉这块内存。这给了我们喘息的机会，我们能有时间处理这个问题。

现在我们可以处理它，有好几个办法。我们可以放弃，就把本关卡停掉，直接回到游戏的主菜单。虽然这样也不是太好，但是至少避免了游戏的崩溃。也可以输入到内存池内部，释放掉一些当前没有使用的内存，抑或是压缩一些内存池的游戏实体，以释放一些内存。

取决于你是怎样处理内存较低的情形，有一个很不错的工具，你可以加到你的游戏里去：内存压力测试模式。在这种模式下，所有可用的内存，除了很小的数量，都可以被分配。这样的模式下，你的游戏能够正常工作下去。如果为使用了动态内存的系统分配添加了一个反馈的机制，那就需要观察一下后果。如果想让游戏更稳定，那就接管所有可以使用的内存，看看游戏能否继续运行。

7.8 结 论

本章提出了内存分配的问题，以及一些处理办法。最简单的办法是只使用静态内存分配。这解决了内存碎片问题，还有动态内存分配的性能开销，但是付出的代价却是失去了灵活性，浪费了很多内存。静态内存分配不能满足这样的情形：动态的世界有着很多的交互性。

动态内存分配就灵活的多了，但是额外的代价是复杂了。我们不得不认真对待关于内存碎片的问题，不得不考虑游戏过程中内存会不会用完，还不得不考虑实际的性能。我们提供了内存管理系统的源代码，可以很容易地集成到已有的代码里去，以跟踪动态内存是怎样分配的，可以提供一些调试上的便利。

最后，我们看到了，使用内存池可以使得我们使用动态内存分配，而没有性能损失或者内存碎片问题。我们也看到，有一些源代码，可以很容易地把任何一个类加到自己的内存池里去。

7.9 阅 读 建 议

下面的书籍有一些有趣的观点，是关于重载 new 和 delete 运算符的。

Meyers,Scott. Effective C++ Second Edition, Addision-Wesley, 1998.

也有一些关于动态内存分配的不错的文章：

Ravenbrook, TheMemory Management Reference, http://www.memorymanagement.org/.

Johnstone,Mark S.,and Paul R. Wilson, "The Memory Fragmentation Problem:Solved?" Proceedings of the Internatinal Symposium on Memory management, ACM Press, 1998.

Hixon,Brian,et al., "Play by Play: Effective Memory Management", Game Developer Magazine, February 2002.

下面的文章是有关内存池的，是挺有趣的文章。

Saladino ,Michael, "The Big Squeeze: Resource Managemant During Your Console Port" Game Developer Magazine, November, 1999.

有一个很不错，解释得也很详细的一个小对象的内存分配系统，可以在下面的网站找到：

Alexandrescu Andrei, Modern C++ Design, Addision-Wesley, 2001.

Boost 内存池的库，作为一个开始，也值得看一眼。

C++ Boost Pool library, http://www.boost.org/libs/pool/doc/index.html.

第 8 章　标准模板库——容器

本章将讲述：

- STL 概述
- 用还是不用 STL
- 序列式容器
- 关联式容器
- 容器适配器

本章并不想讲述关于 C++标准模板库所有内容，因为这个话题需要用一本书的篇幅来讲述。幸运的是，已经有人写了这本书，如果你过去没有用过 STL（标准模板库），可以找一下本章末尾的阅读建议所说的 STL 书籍，然后再回过头来继续看这本书。

本书的以下两章将集中讲述怎样更有效地使用 STL，对于游戏开发或者其他一些对性能要求较高的编程，读者需要了解一些基本概念。比如，Vector 和 list 的不同，还有如何使用 Iterator 来遍历一个容器中的所有元素。虽然，STL 用的越多，读者就越清楚所要解决的问题，也就越会赞许使用 STL 的方案。但只要了解了刚才提到的基本概念，读者就可以学习本章的内容了。

本章将首先问我们自己是否需要在项目中使用 STL，接着会详细了解容器的主要类型，熟悉它们的性能和内存使用情况。这些因素将在游戏开发过程中变得很重要。第 9 章将讲述算法和一些高级话题，比如可定制的内存分配算符。

8.1　STL 概述

本节是 STL 基本概念的快速回顾。如果从未用过 STL，那最好找找本章末尾的阅读建议里提到的书先看看，获得一些实际的经验。还有，应当对模板有一些入门的认识（参见本书第 4 章关于模板的介绍）。本章也会复习这些基本的内容。

C++标准模板库是类的集合，它提供了容器来存储不同类型的结构，也提供了 Iterators（迭代器）来访问容器里的元素，提供了算法来对容器执行操作。所有的类都是基本的，所以它们可以被改造成可以供我们自己使用的类。

有好几个不同类型的容器，每一个容器都有着各自不同的操作，不同的内存使用情况和性能特性。为自己的工作选择合适的容器是很重要的环节。两类主要的容器是 sequence 类容器（序列式容器）和 associative 类容器（关联式容器）。所有存放在序列式容器里的

元素是有顺序的，而对于关联式容器来说，则不理会元素的存储顺序。

```
// Adding three elements to a vector of integers
vector <int> entityUID;
entityUID.push_back(entity1.GetUID());
entityUID.push_back(entity2.GetUID());
entityUID.push_back(entity3.GetUID());
```

Iterators 使得我们可以访问容器里的每一个元素。所有的容器有两个非常重要的函数，这两个函数分别返回两个不同的 Iterators：begin()函数返回第一个元素的 Iterators，end()函数返回最后一个元素后面的那个元素的 Iterators。注意 end()函数返回的 Iterators 并不是最后一个元素的，而是它后边的那个元素的。这是整个 STL 里非常重要的约定。大多数与元素范围有关的的函数都需要传递 Iterators 范围里的第一个元素，传递 Iterators 范围里的最后一个元素。这初看起来好像有点奇怪，但是当需要遍历一个容器的所有元素的时候，或者实现一个算法的时候，这样的约定会使事情简化很多。

一旦有了容器的一部分的 iterator，就有可能使用它，作为起始点来获取它附近的元素，至少访问它的下一个元素。不是所有的 iterator 拥有同样数量的函数功能实现，但是有时候有可能获取前一个元素，或者随机的元素，这一切取决于 iterator 的种类。

```
// Traverse the vector of UIDs
vector <int>::iterator it;
for ( it = entityUID.begin(); it != entityUID.end(); ++it ) {
        int UID = *it;
        // Do something with the UID
}
```

STL 提供了一整套标准的算法，这些算法可以应用到容器和 iterators。它们都是相当基本的算法，可以把它与代码结合起来，以得到想要的结果。这些基本的算法有：在容器里查找元素，复制元素，保存元素，排序等。所有的 STL 代码和算法都经过了高度的优化，它们都使用了最可行、效率最高的实现，而不使用那些看起来虽然简单，但是用起来却很慢的算法。

```
// Reverse the entire vector of UIDs
reverse ( entityUID.begin() , entityUID.end() );
```

STL 所提供的所有的类和函数都被命名空间 std 所封装。这意味着如果想使用 STL，需要在所有的 STL 名称前加上 std::或者在想使用 STL 的.cpp 文件的头部加上 using namespace std 这样的语句。

8.2　用还是不用 STL

当开始一个新项目的时候，程序员应当问问自己，是否需要在项目中使用 STL。你可能会想自己知道这个问题的答案。尽管本书有两章的篇幅专门讲述 STL，我们还是先一次性地通读一下 STL，争取给出一个合理的答案。如果用过 STL，而现在是 STL 的忠实使用者，那就跳过这部分直接阅读本章的精华吧。

8.2.1　代码复用

使用 STL 最大的一个争论是很明显的，STL 是一大堆代码的总体，而且是别人为你写好的。人们已经实现了一些不同种类的容器和算法，而且把 bug 查的差不多了，也移植到你使用的平台上了，你也可以免费使用或者仅需为此支付少量的银子，价格绝对公道。

大多数 STL 的代码可以用来为游戏或者一些工具软件搭建基本的功能块，比如链表、哈希表、排序算法，还有查找算法。你真的想花很多时间来写这些代码吗？因为 STL 多是由模板和模板化的函数所构成，所以把 STL 集成到代码库里是有可能的。你没有必要为改写你所有的链表元素必须从某个基类继承，或者其他的什么改动。

STL 代码库已经被调试很多年了，世界上有许许多多的项目已经使用它了，这当然也包括一些游戏项目，效果也还不错。换句话说，STL 将要成为很稳定的代码了。如果使用 STL 的时候碰到了什么问题，那一定有别人早就碰到这样的问题了，于是也就有人解决这样的问题了。网上有一些活跃的社区，里面有大量的资源，在那里还可以找到你所碰到的问题的答案。

你也许获得了一些源代码，很不幸的是，这些代码不像听起来的那么有用，因为这些代码对一般的读者来说，不太容易理解。高度个性化的代码像缠结不清的鸟巢，优化的模板代码，到处是文件包含（include）和宏定义（define）。别指望能看到一个简明的小函数来说明 vector 是怎样工作的。相反，你能看到好几种实现，这取决于使用的平台和当前的配置情况。为解决一个特定的问题，上网浏览论坛上的在线讨论，常常比钻研那些成千上万行的代码来的更快一些。想要改动这些代码，只能是令人畏缩的想法。但是假如事情变得很糟，你倒是可以看看源代码，看看它究竟是怎么实现的。

STL 不能忽视的另外一个好处是，世界上有很多人在使用它。使用 STL 的新手，如果没有必要学习新的一套内部开发使用的类，比如链表或者排序算法，那他会觉得 STL 速度很快。此外，有些库设计成与 STL 交互的接口，这使得 STL 变得更加让人容易接受了。

8.2.2　性能

现在该谈谈给大多数程序员留下深刻印象的性能问题了。游戏程序员特别在意性能问

题，差不多就是这样。可是 STL 能提供好的性能吗？

答案当然是 Yes。注意 STL 已经被许许多多的人使用很多年了，有一些非常聪明的人承担起了优化 STL 的责任。所以，在每一个平台上，从某个特定的容器或者算法中能榨取的性能都差不多到极点了，而这一切都是透明的。

如果你不太相信这样的 STL 代码的性能超过你自己写的程序，那就编译 STL，试试看吧。通常，你会发现 STL 的性能超过你的程序一大截。即使你的代码你花了好几天或者好几周的时间来优化，STL 也不会比你的代码慢。

但是超过 STL 的性能也并非不可能的。通常，这需要开发者对要操作的数据有着非常深刻的了解，或者对工作的平台了如指掌。平台的差异，STL 是不会了解的（或许因为 STL 对于移植到这个平台太陌生，也没有时间来充分考虑）。有多少场合你需要那多出来的一点点的性能呢？通常不会太多吧，至少不会在整个项目中的过程中都需要。如果真的需要，可以用自己写的函数替代那些特别慢的代码。到了这个程度，才能用 STL 写出更健壮、更快速的程序来。

需要考虑的一件事是，STL 到底有多强大？STL 由于拥有一整套基本的数据结构和算法，应用非常广泛。而且使用 STL 问题中出现的基本元素不再是指针，不再是内存缓冲区，而是链表的节点或者集合的入口等这样更为自然的方式，所以使用 STL 的方案往往更合适一些，更好一些。

也许你仍然关注于 STL 的性能问题，那就记住：使用 STL，可以使设计的算法更有效率，因为 STL 使用了高级的数据结构和算法。使用比较底层的办法来解决同样的问题，虽然更简单，不过性能却差很多，因为这样的代码太容易写了。使用 STL 可以把一些元素送入一个哈希表，这样以后查找元素的时候，消耗的时间是常数级的（时间复杂度为 O(1)）。不使用 STL，大多数程序员会简单地把这些元素放到一个固定大小的数组里，搜索消耗的时间是线性的（与元素的个数成正比，时间复杂度为 O（N）），而且还要时刻记得数组不要越界。

虽然 STL 可以提供很不错的性能，但是如果使用不正确的话，也能使程序的速度显著地降低。由于 STL 的复杂性，要求程序员也要有责任。有时候，很简单的问题，比如选择错了数据结构，可能就会成为性能杀手。还有一些更细微的问题，这包括为一个特定种类的容器动态内存分配，或者额外的复制（由于复制元素而引入）也会带来很大的影响。为了更有效地使用 STL，特别是对性能要求比较高的环境，比如游戏，理解 STL 背后所做的事情以及为合适的工作去选择合适的工具，就显得非常重要了。

8.2.3 缺点

STL 就没有一点不好的地方了吗？有没有什么原因，使我们不能在所有的游戏和工具的开发中使用 STL 呢？很不幸，还真有几个原因。

STL 最显著的缺点是，有些时候调试很困难。你应该忘记可以以单步调试的方式进入

容器操作或者算法的内部。STL 过度地使用模板，这使得编译器想要交互式地调试、设置断点变得非常困难（或者不可能）。但是这不算太糟；毕竟，STL 的代码已经被别人调试得很好了，我们没有必要再把手弄脏一次。主要的问题在于查看创建的容器的内容。一般情况下不可能看出一个 iterator 指向的是什么，或者查看一个 vector 里的所有元素，这么简单的事情都不行。不过也有可能，强制编译器这么做就可以了。

使用 STL 的时候，编译器也许会产生一些错误信息，这些错误信息也许不是你要处理的。它们一般很长，有很多行，都是含义模糊的信息，这些错误经常指向压根就不是我们写的代码，仅仅是因为为容器指定的类的类型不对。要想正确地分析这些提示信息，需要经验和耐心，只有这样才能找到真正的 bug 所在。

STL 的最后一个缺点是关于内存分配的。STL 的本意是一个有着很大内存的通用性的计算环境。用 STL 开发工具软件是最常见的应用，与此同时，在特定的游戏平台上，尤其是一些游戏终端，却不是很适合使用 STL。幸运的是，可以为特定的容器提供定制的内存分配，可以自由控制分配内存的时机。本章的后面详细地讲解了内存分配的问题。把 STL 集成到自己的内存管理模块中，需要做一些额外的工作。要么提供更多的内存分配函数，要么修改一些全局的内存分配函数成为使用我们的内存管理系统。幸运的是，开发可以从一个游戏开始，而不必要去考虑刚才所说的两个方面。只有在非常必要的情况下，才需要考虑比较底层的内存细节问题。

总的来说，不要在所有的开发中使用 STL，这也没有什么好的理由。在开发的游戏中是否使用 STL，你应当好好考虑考虑。如果使用 STL，你就得修改自己的函数了。有很多 PC 和终端上的游戏已经移植成使用 STL 了，它们已经很充分地体会到 STL 所能提供的强大功能了。

8.3　序列式容器

之所以叫序列式容器，是因为它们最显著的特点就是它们所保存的元素是有顺序的。正是由于元素是有顺序的，才使得在特定的位置插入或者删除元素（比如在头部、尾部或者中间的任何位置）变得有可能。序列式容器主要有 vector、deque、list。

8.3.1　vector

也许 vector 是最简单的容器，但也是应用最多的一种容器。可以在序列的任何位置添加或者删除元素。接下来也能看到由于位置的不同，添加操作和删除操作的性能也不一样。vector 提供了双向的 iterator，也就是说，通过一个 iterator，可以前向或者后向访问它的邻居。此外，一个 vector 的元素可以以随机的顺序访问，像 C 语言里的数组那样：

```
vector <int> entityUID;
```

```
    //Fill it up
    if ( entityUID[5] == UID )    //…
```

除了这一点像数组之外，vector 不会限制加入其中的元素的个数。使用数组的时候，最基本的技巧就是需要大致猜出最多用多少个元素，然后为这样的数组分配上界。只要为这个数组添加一个元素，首先需要确定没有越界。如果数组已满，好的情形是能得到一个错误的警告，乐观一点，程序能正常工作。如果碰上最糟糕的情形，程序会尝试为这个已经满了的数组增加一个元素，这会导致内存异常或者内存出现页面错误，然后程序就崩溃了。扔掉数组吧，使用 vector 就能避免这些问题。

1. 典型实现

vector 把它所有的元素连续地放在一个比较大的内存块里。这个内存块为了以后能添加新元素，通常都额外多一些空间。每一个元素都被复制到正确的位置，不需要添加任何指针或者其他附加的信息（除了一些“废话”，这与具体平台有关系）。由于 vector 所有的元素都是一个类型，大小也一样，所以它在 vector 中的位置可以依据它在 vector 中的索引而计算出来。

除了要一块内存来保存元素，它还需要一个指针指向元素的头部，还有其他一些信息，诸如当前的元素的个数以及当前已分配的内存块的大小。图 8.1 展示了一个 vector 容器的典型实现。

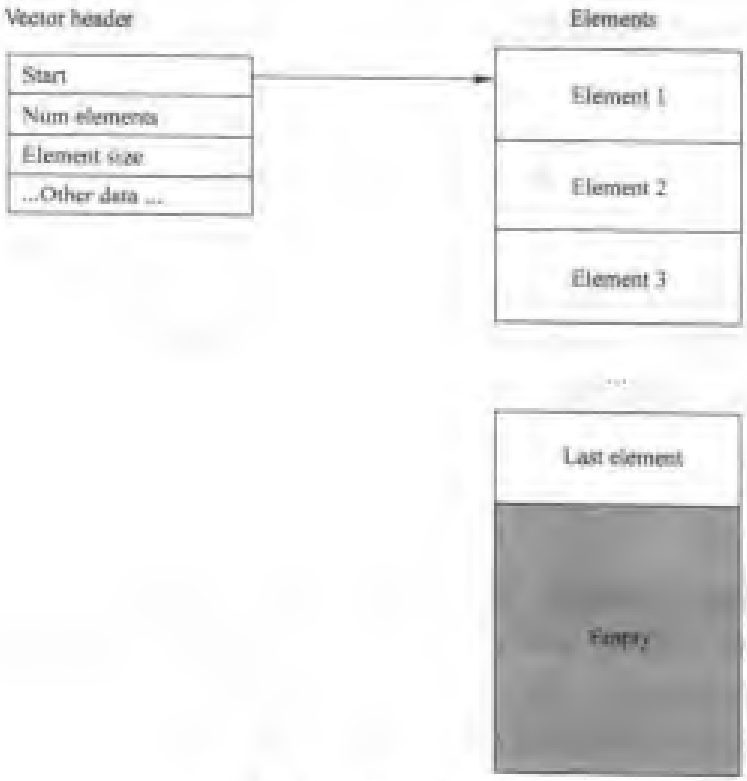


图 8.1　vector 的实现

2. 性能

在 vector 的尾部添加或者删除元素是非常有效率的，就像数组一样，STL 的有关文档可以保证它的性能开销为 O(1)。它告诉我们的是，插入操作消耗的时间不是取决于 vector 的大小。但是究竟能有多快呢？插入一个新元素所花费的时间如果超过 30ms，那就没有什么意义了。幸运的是，STL 太快了，所花费的时间就是用于把元素复制到它的位置上。

然而，如果插入或者删除的元素不在 vector 的尾部，而是其他位置，那这个操作的开销可就大了。这是因为需要复制从插入点到尾部的所有元素，然后逐个的移动。这并不是说，插入或者删除一个中间的元素是不可能的。vector 支持这样的操作，只是没有那么高的效率而已。

此外，在 vector 的尾部插入一个元素可能会引起内存块用完其内存，这会导致内存的重新分配，需要复制所有的元素，我们将在内存使用那一节里看到这一点。小心一点，就完全可以避免在需要很高性能的循环中使用它。因此考虑 vector 的总体性能的时候，就不要考虑这一点了。

如果程序从不同的位置增加或者删除元素也不是太频繁，或者容器里的元素的个数不是很多的情况下，vector 容器可能就是最好的选择了。

遍历容器里的所有的元素速度也非常快。遍历一个 vector（只要代码写得好）可以像遍历一个数组一样高效，大概的代码是这样的：

```
vector <int>::iterator it;
for( it = entityUID.begin(); it != entity.UID.end(); ++ it ) {
   int UID = *it ;
   //  Do something with the UID
}
```

3. 内存使用

我们前面就知道了，vector 把所有的元素保存在一个较大的内存块里。这个内存块不仅要大到能存放目前所有的元素，还要多出一些空间以存放将来要插入的元素。随着元素的插入，这块多出的空间也越来越小，直到插入一个元素后，所有的空间都用完了。这个时候，vector 就会重新分配一个更大的内存块，把已有的元素都复制过去，再添加那个刚要插入的元素，然后再释放掉那块老内存。通常，新的内存块将是老的那块内存的两倍。这样多的空间可以用 vector 来添加新的元素，而无需频繁的分配新内存，或者分配太多的内存导致很多浪费。

注意，没有提 vector 的大小不断缩小的事情。vector 自己从来不会变小，所以，如果一个 vector 通过不断的添加记录已经变得很大，然后又删除大多数元素，那将导致有一些已分配的内存没有用上。

聪明的读者也许已经意识到，刚才说的问题有值得探讨的地方。一旦 vector 达到了大小的限制点，就会分配新的内存，而且所有的元素会被复制到新的内存里去。那么，如果

有一个指针指向元素，会发生什么呢？运气就没有那么好了。vector 从来不会保证指针的有效性，也不会保证在插入元素后能够引用。如果要删除或者要插入的元素的位置在 vector 的中间，就不得不把这个元素之后的所有元素前移动（或者后移）一个位置。如果要引用某个特定的元素，甚至在这样的操作之后，应当保存它们的索引，然后通过[]操作函数来访问它（如 one_vector[2]来引用 vector 里的一个元素）。

分配一块新的内存，复制所有的元素到新的位置，这也许不会太轻松，我指的是性能开销。vector 里的元素越多，那复制所有元素花费的代价就越大，程序的性能也就越糟。STL 有一些方面需要我们小心的吗？是的，也许是这样。幸运的是可以采取一些措施来减轻这种负面的影响。

第一个方面就是为 vector 预先分配足够多的内存，这样可以避免频繁地分配内存。记住每次 vector 内存用光的时候，它的内存会加倍。所以，如果最先分配的内存为 16MB，那它就会逐渐增为 32MB、64MB、128MB 等。如果知道需要至少 800 个元素，则需要 6 次分配内存（和复制）。通过使用 reserve()函数，并预先分配 1024 个元素的空间，可以避开这个问题。但是 reserve()函数不会改变 vector 的元素的个数（这和 resize()函数一样），它仅仅是改变为当前所有的元素和未来的元素所使用的内存空间的大小。

vector 可能在运算的时候放满了元素，当运算结束后就丢弃掉。如果频繁发生的话，就意味着我们不断地为 vector 分配内存、释放内存（如果没有使用 reserve()，那可能会发生很多次）。这个问题的一个较好的解决办法是，保存这个 vector，在使用之前清掉它所有的元素就可以了。清掉这些元素不会释放任何内存；它只是清空 vector，并为每一个元素调用析构函数（假设这些元素有析构函数可以调用的话）。

STL 的一个很方便的特性就是它所有的元素是连续地存放在内存里的；也就是说，只有一块内存块，元素是一个挨着一个保存在这内存块里的。这带来以下好处：第一，遍历所有的元素能使得数据缓存的数据保持一致性。这样的特点对一些平台来说更重要，但是在元素上千的情况下，即使在 PC 平台上，这也很明显；第二，连续排列的特点，使得可以通过向函数传递一个指针，一次性地把 vector 的内容直接传递给函数。这是一个简单的优化，可以作为不使用 STL 的函数的接口。例如，一个指向 vector 内容的指针，可以作为顶点数据数组传递给 3D API 的函数。这样使用的时候，需要谨慎一点，确信这样的函数不会增加或者删除元素，要不然 vector 就会和内容不同步了。

4. 建议

在尽可能多的地方使用 vector，vector 的性能很不错。除非内存使用太紧张了，甚至于连一个 vector 所需的那多出来的一点点字节数都支付不起，否则就不要再使用数组了。即使提前知道了元素的最大个数，像这种情况都没有必要再使用数组了。

在尽可能多的地方使用 vector 可以给你一些灵活性，没有必要再维护元素的个数这类问题了，而且它和游戏中用到的其他 vector 使用的是同样的接口。

vector 对于高性能的程序来说，有着很多完美的特性。因此，应当学会考虑 vector 是否适合某个特定的程序，只有在不能满足的情况下再考虑其他比较复杂的容器（如表 8.1

所示）。在游戏开发方面，vector 比较适合的场合有：

- 玩家可以循环选择的武器列表
- 游戏中当前的所有玩家列表
- 一些预先分配的缓冲区以供接收网络消息
- 有着简化的几何形体的顶点列表，将来用做碰撞检测
- 一颗树的某个结点的所有孩子（如果孩子是静态的或者孩子数不多）
- 一个游戏实体（entity）所有可能玩的动画列表
- 一个游戏实体（entity）容纳的所有部件列表
- 沿镜头运动的路径的点列表或者 AI 路径的计算结果

表 8.1　vector 总结表

在尾部插入/删除元素	O(1)
在头部插入/删除元素	O(N)
在中间插入/删除元素	O(N)
内存分配	很少，仅仅增大
遍历性能	最快（像 C 语言里的数组）
是否支持连续的内存访问	是
iterator 有效性验证	在插入或者删除之后
内存开销	12~16 字节

8.3.2　deque

对于那些不熟悉这种数据结构的读者来说，它的发音和“deck”一样，代表的意思是“双向列表”（double-ended queue）。

deque 几乎等同于 vector，但是性能上有一点差别。像 vector 一样，deque 能随机访问其中的元素，也可以在队列的任何位置插入和删除元素，虽然开销有所不同。deque 与众不同的地方在于，不管在头部还是在尾部，它能提供快速的插入和删除操作。

1. 典型实现

双向队列类似于 vector，它的所有元素也保存在很大的内存块里。然而，主要的差异在于它有着很多个内存块，而不是仅仅一个。随着越来越多的元素插入到队列里（无论插到头部还是尾部），会分配一个新的内存块，新插入的元素就放在那里。不像 vector，这里没有必要复制现有元素到新的块里面，因为老的内存块依然有效。

为了记住这些内存块，双向队列有很多指针指向了每一个块。使用的内存块越多，需要的指针就越多。可以把这么多指针理解为一个 vector，这个 vector 保存着很多指针，每一个指针指向一个内存块。双向队列的头部还有其他一些信息，诸如：元素的个数、队列

的第一个元素的指针、队列的最后一个元素的指针等。图 8.2 展示了 deque 的一个典型实现。

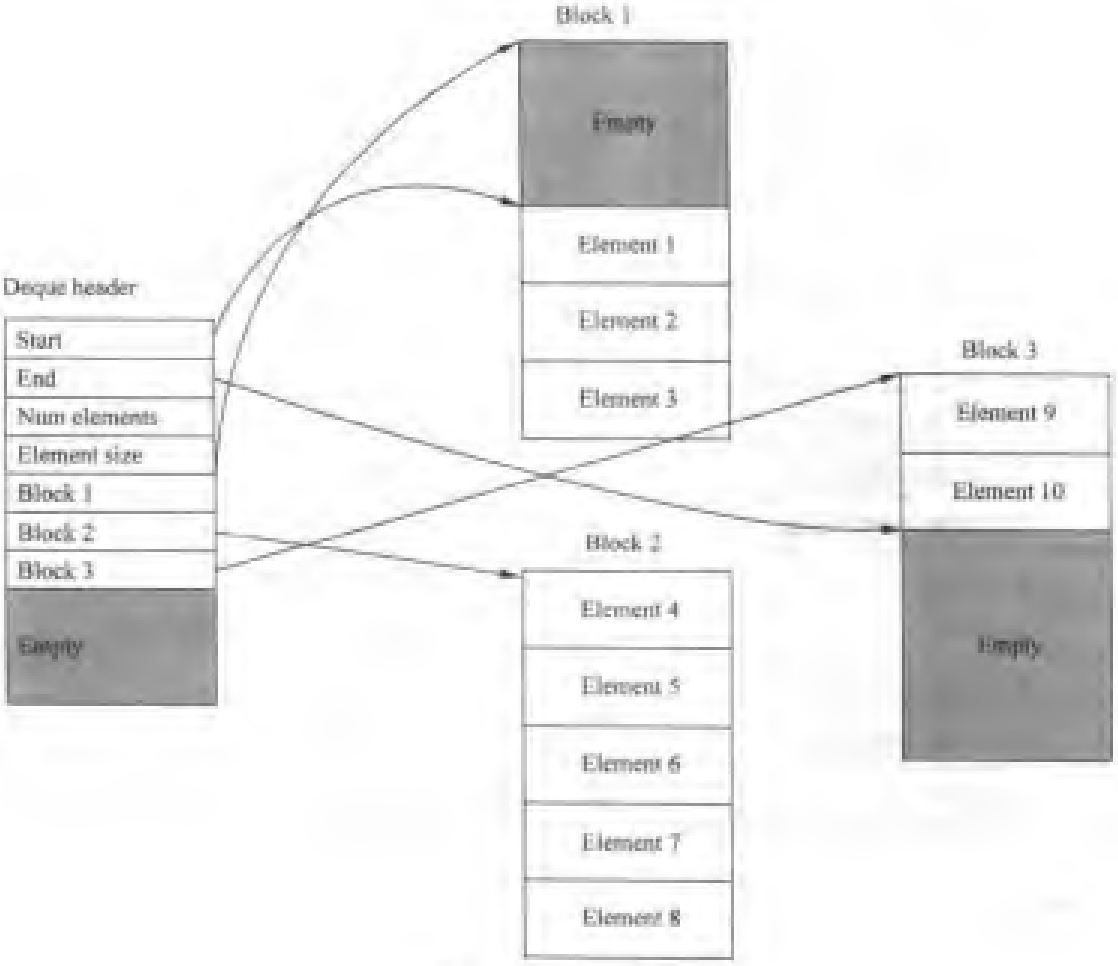


图 8.2 deque 的实现

2. 性能

对 vector 来说，在尾部插入和删除一个元素是很快的操作（时间复杂度为 O(1)）。此外，和 vector 也一样，在中部插入和删除一个元素相对来说就慢得多（时间复杂度为 O(N)）。

然而，正如先前所说，deque 和 vector 的主要区别在于，在头部插入和删除一个元素与在尾部进行操作的情况一样。这使得 deque 成为一个完美的数据结构，可以实现 FIFO（先进先出）的特性，像队列那样。

一个 deque 通常把它的元素保存在几个内存块中，不像 vector 那样只用一个内存块。在每一个块里，元素是顺序排列的，所以这个块也可以获得数据缓存一致性的好处。然而，当要访问的元素位于另外一个内存块的时候，性能会略有降低。即使如此，由于这些块相对来说都很大，遍历所有元素并不比 vector 来得慢。其余的操作通常比 vector 慢一点或者最多和 vector 一样快。

3. 内存使用

在 deque 的使用过程中，会不断地发生内存分配的情况。这是很显然的，如果继续向 deque 插入元素就需要新的内存块。类似地，删除元素最终也会导致一些内存块被释放掉。

即使 deque 里的元素的个数保持相对不变，deque 也会不断地分配内存。如果对象在尾部添加，在头部删除，新的内存块会添加在尾部，由于删除已经变空的那个内存块会在头部被删除。在某种程度上，可以这样理解：deque 里的元素数量保持不变，deque 在内存里做的是“滑行运动”。

由于 deque 的动态性，所以无法为固定元素数量的 deque 预先分配内存。STL 就没有像 vector 里那样，为 deque 提供 reserve()这样的函数。

关于 deque 的最后一个重要问题，它可能分配一个“很大”的初始内存块。在一些 STL 的实现里，这个初始内存块可能高达 4KB。如果需要同时创建非常多的 deque 时，一定要记住这个问题。

4. 建议

由于缺乏有效的内存控制，以及难以预料的分配后果，deque 对于那些内存预算紧张，或者内存分配代价很大的场合，可能不是一个很好的选择。如果要处理的元素不多，vector 是不错的选择。即使每一次删除头部的元素，都需要支付额外复制的代价。要不然，可以考虑 list，list 可以定制内存分配。

deque 的用途不像 vector 那么多（如表 8.2 所示）。即使如此，在游戏开发过程中，还是有很多场合适合使用 deque 的：

- 保存游戏中的消息放到队列，然后以先进先出的顺序来处理。
- 以宽度优先的原则，遍历一个树形的对象体系，遍历的顺序可以保存在一个 deque 里。

表 8.2　deque 总结表

在尾部插入/删除元素	O(1)
在头部插入/删除元素	O(1)
在中间插入/删除元素	O(N)
内存分配	有周期性，一般用途
遍历性能	几乎和 vector 一样快
是否支持连续的内存访问	差不多；一些连续的内存块
iterator 有效性验证	在插入或者删除之后
内存开销	头部多于 16 字节；初始内存块可达 4KB

8.3.3 list

list 是另外一种类型的容器，但是它和 vector 和 deque 有着显著的不同。list 提供了双向的 iterator，这意味着只要给出 list 中某个特定元素的 iterator，就可以访问它的下一个或者上一个元素。然而，list 没有像 vector 和 deque 那样，提供随机访问元素的能力。任何需要随机访问元素的算法都不要采用 list。我们损失了灵活性，却得到了性能和对一些特定操作比较方便的好处。和 vector 和 deque 一样，可以在 list 的任何位置插入、删除元素，只要有这个位置的 iterator。

1. 典型实现

list 被实现为一个有着双向链接的元素列表。和 vector、deque 不同的是，元素是以结点的方式保存的，而不是像 vector 和 deque 那样存放在一个较大的内存块里。每一个结点都有一个前向指针和一个后向指针。

列表也有通常的头部信息，有一个指针指向第一个元素，有一个指针指向最后一个元素，也许还有其他一些信息。一个典型的 list 实现如图 8.3 所示。

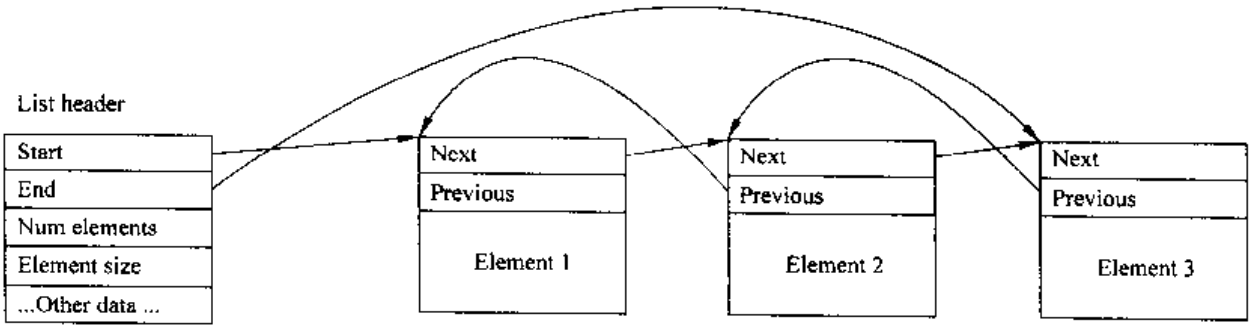


图 8.3 list 的实现

2. 性能

你可能猜到了，list 最大的好处就是，在它的中部插入或者删除元素不需要付出什么代价。任何插入或者删除操作，不管在什么位置，时间消耗都是常量（与要插入或者删除的位置无关）。因为要做的仅仅是改变指针的指向。当然，需要有一个 itertor 指向要执行插入、删除操作的位置。得到这个 itertor 消耗的时间，可能就不是常量了。

遍历 list 中所有的元素比 vector 慢得多。为了遍历一个 list，程序需要读出每一个结点的地址，接着再访问内存中的另外一个位置，这使得数据缓存的内容一致性变差了，于是遍历变得很慢，其速度和 vector 相比差一个数量级。不过，别着急。记住遍历的时间可能不是性能最大的瓶颈。所以，即使遍历的时间很长，最终不一定很明显。因为最终性能取决于特定程序。

涉及到重新排列 list 的元素的操作（例如排序）可能效率会较高，这是因为不需要把

元素复制来复制去，只要修改它们的指针就可以了。元素越多，复制元素的开销越大，如果使用 list，就会获益更多。

3. 内存使用

list 与其他序列式类型的容器完全不同。其他的容器往往是分配一块大内存，尽可能多的往里塞元素。list 则是为每一个元素分配很小的内存块，这些小块的内存就是 list 的一个一个的结点。

在插入或者删除 list 位于中间位置的一个元素时，不会妨碍它周围的元素，这是 list 的一个优点。这也意味着需要保存每一个元素的指针，即使在做插入或者删除操作的时候。这样的话，可以使很多算法的实现变得很方便。

排列 list 的元素，会导致几乎每一步都会分配内存，这是一个缺陷。如果某平台上动态分配内存速度很慢，这就会严重影响性能。不过如果稍微费点劲，使用自己定制的内存管理器，可以把这个问题的影响限制在很小的程度上。

结点在内存里不是连续排列的。它们是在程序运行的过程中，在不同的时刻动态分配的，所以这些结点会散布在整个内存空间的各处。由于这样的特点，数据缓存的一致性就会比较差，这会对性能带来严重的影响。幸运的是，可以改进这个数据缓存的一致性，只要使用前面介绍的定制的内存管理器。自己定制的内存管理器不仅能避免通常的动态内存分配，还能使结点尽可能地挨在一起，从而提高 list 的性能。但是使用自己定制的内存管理器并不能完全使这些元素挨在一起，在最糟糕的情况下，即使使用自己定制的内存管理器，元素的位置也没有什么变化，还是没有一点位置上的连贯性。

list 的另外一个缺点就是额外的内存消耗。一个典型的 list 结点会额外需要 8~12 字节的内存。这对 PC 平台来说不是什么问题，但是对于那些有着内存限制的游戏终端来说，如果我们有很多这样的小结点，那么这额外的内存消耗加起来就是很大的数字了。

4. 建议

list 是一个通用的容器。在自己的程序里完全使用 list 而放弃 vector 和 deque 是一件很诱人的事情。然而，list 的性能很差，这使得它不适合高性能的场合。只有在已经确定 vector 和 deque 不适合解决问题的场合下，才考虑使用 list。

有趣的是，有一些 STL 的实现提供了另外一个不同的容器：slist，虽然这还不是标准。slist 和 list 很像，但是它只是一个单向链表。这样每一个结点就少用了一些内存（不需要后向指针了），指针管理的内容也少了一些，但是不能从一个结点后退以引用前一个结点。所以 slist 提供的 iterator 仅仅是前向的 iterator。还有，slist 有一些不可预料的性能特点，因为有一些操作依赖于能够访问前一个结点（比如在 slist 中删除一个结点）。所以最好在内存节省非常重要的情形下使用 slist。要不然，一般的 list 才是更好的选择。list 有很多用途，可以应用在游戏开发方面的有下列一些（如表 8.3 所示）。

- 一个链表用来保存游戏中所有的游戏实体，在游戏过程中，会有大量的添加和删除操作。

- 一个链表用来保存 AI 的评估结果，其中有一些会被选中而被删除。
- 一个链表用来保存每一帧里所有的 mesh，并以材质和渲染的状态排序。

表 8.3 list 总结表

在尾部插入/删除元素	O(1)
在头部插入/删除元素	O(1)
在中间插入/删除元素	O(1)
内存分配	每一次的插入和删除操作
遍历性能	比 vector 慢得多
是否支持连续的内存访问	否
iterator 有效性验证	从来不做
内存开销	8~12 字节的头部信息，8~12 字节每结点

8.4 关联式容器

序列式容器保存了插入或者删除操作的元素的相对位置。关联式容器则放弃了这个特性，把注意力放到如何尽可能快的把一个元素从容器里找出来。

对于一个序列式容器来说，查找其中的一个特定的元素时间复杂度为 O(N)，因为需要查看容器中的每一个元素。对于 vector 或者 deque 的情况来说，如果元素已经排序了，使用二分法查找的时间复杂度为 O(ln N)，因为二分法可以快速的随机访问数据。然而，为了利用这个便利性，必须为元素排序。这并不是我们所期望的，而且当插入新元素或者删除已有元素的时候，维护已有元素的顺序也很费劲。

关联式的容器提供了时间复杂度为 O(ln N)，最差为 O(1)的搜索效率。通常情况下，为了可以快速定位元素，每一个元素都有一个键来表明自己的“身份”。有时候，元素本身就是它们的键。

8.4.1 set 和 multiset

set 容器（集合）和数学中的集合概念很相似：它包含一些对象，但是没有一点关于顺序的信息。元素与集合的关系只有一个：要么在集合里，要么不在，没有第三种可能。

应该可以检查要加进集合的两个元素是否相同。特别是，应该定义运算符“<”，或者应该提供一个函数，该函数作为模板的一部分，来比较两个对象是否相同。

调用函数 insert()就可以为集合加入新的对象，调用 erase()函数可以删除某个对象。要查找某个特定的元素是否在集合里，调用 find()就可以了，算法的时间复杂度为 O(ln N)。定义集合的模板为：

```
set<Key, Compare, Alloc>
```

Key 是要为集合加入的元素的种类，Compare 是当为集合插入元素时，用来比较元素是否相等的函数，最后一个参数 Alloc，将在第 9 章 STL 算法及高级话题里介绍它。通常，不是所有的参数是必需的，所以模板为它提供一个默认的 allocator 和一个默认的比较函数。特别是，还会尝试使用运算符<。所以惟一要做的是，如果元素没有定义运算符“<”，或者假如想要做不同类型的排序，为之提供一个比较函数。

```
set<int> objectives;
objectives.insert( getObjectiveUID() );
//…

if ( objectives.find( objectiveUID ) != objectives.end() ) {
   //  That objectives was complted.Do something
}
```

集合仅为相同的元素保存一个备份，所以多次插入同样的对象并不会改变集合。multiset 和集合不同，如果多次插入同样的元素，它就可以保存该对象的多个备份。multiset 也有一些自己特定的函数，比如 count()函数，可以返回某个特定的对象对 multiset 中出现了多少次。

如果想在一批数据里面剔除重复的个体，集合可能是最好的容器了。要做的仅仅是把这些数据一个一个的加入到集合里，然后再把集合的每一个元素读取出来就可以了。这个方法的效率很高。当元素的个数增长很快的时候，O（N^2）的时间复杂度比 list 中查找重要的元素让人难以接受。

为了创建更有意思的程序，可以把自己的比较函数传给集合。设想一下，有了一些 3D 空间的点，还想挑出所有在一定距离之内的点。一个快速的方法可能是创建一个相等函数来比较两个点，但是只考虑两个在最小距离之内的两个点。可以创建一个点的集合，这个集合使用比较元素是否相等的函数。把所有的点都放进去，接着再把它们读取出来。

```
struct PointNearbyLess {
   bool operator() (const Point3d & pt1,
                    const Point3d & pt2) const
{
   float fDist = ::Distance(pt1, pt2);
   return ( fDist > POINT_COMPARE_THRESHOLD );
}

typedef set<Point3d,PointNearbyLess> PointCollapseSet;
PointCollapseSet pointSet;
// Insert points in pointSet…
```

注意，并没有真正创建一个比较是否相等的函数，而是创建了一个不相等的函数，或者更特殊一点，一个“小于”函数。这是因为 STL 使用这个“小于”函数，而不是相等函数，在集合的实现中，不使用“==”，而是使用：

```
!(k1 < k2 ) && !( k2 < k1)
```

以后使用 STL 的时候，会比较频繁地使用“小于”函数。

集合和 multiset 被称为排序的关联式容器，因为除了当做关联式容器之外，它们保存的元素是有顺序的（由比较函数来确定它们的顺序）。这意味着，除了可以非常快地找出任何元素外，还可以以这样的顺序遍历所有的元素，不是所有的关联式容器都有这个特性的。

1. 典型实现

由于容器越来越复杂，不同版本的 STL 对于这些容器的实现差异也很大。这里所描述的仅仅是 STL 的典型实现，我们就是以 STL 的典型实现作为关于性能和内存使用的讨论基础的。如果使用的是一个比较晦涩的 STL 实现，或者在一个不熟悉的平台，可能需要查看源代码来检验实现的细节是否大致符合这样的思路。

集合和 multiset 通常实现为平衡二叉树的形式，搜索的时间复杂度为 O(ln N)。每一个元素在自己的结点里，和 list 的情况一样。但是在这里，这些结点组织成二叉树的形式，而且它们还使用比较函数（该函数由模板提供）来确定自己的顺序。集合的一个可能的实现如图 8.4 所示。

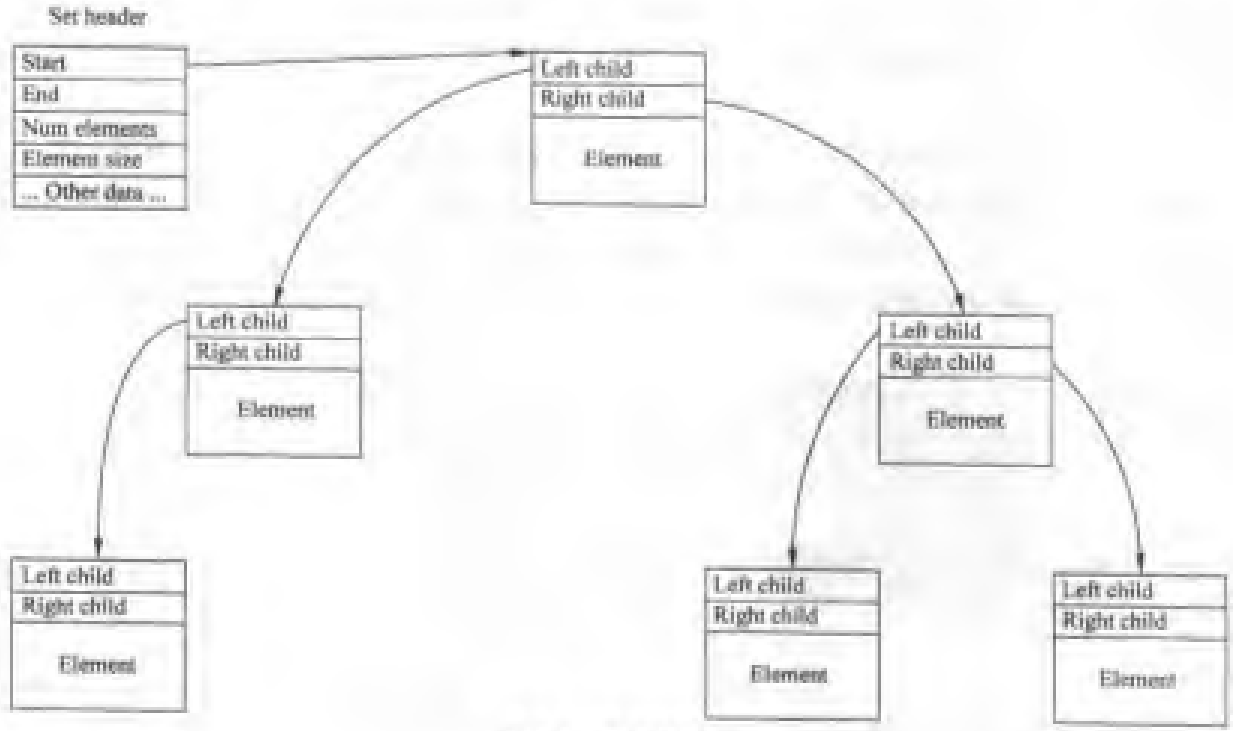


图 8.4 集合的实现

2. 性能

集合的强大的功能在于，我们可以快速找出一个元素是否在一个集合里。有多快呢？集合通常实现为平衡二叉树的形式，因而比较次数的复杂度为 O(ln N)。我们不得不考虑，这些比较的实际支出为多少。比较整数就很简单、很直观，但是比较两个很大的矩阵或者字符串，那开支可就大了。

很明显，集合在元素数量很大的情况下，可以得到很多的实惠。如果元素数量不多，使用集合也很方便。但这种情况下，如果使用 vector 查找也很快。

另外一个替换的办法，尤其是当集合里的内容变化不是特别频繁的时候，就是把元素用 vector 来保存，为它们排序，然后使用二分法查找算法来判断某一个元素到底是不是在这个 vector 里。最开始还是有点开销的，比如初始化时把元素插到 vector 里，以及排序整个 vector，但是如果以后 vector 的内容不会改变，而且主要进行的是查询操作的话，那这点开销还是很划算的。

插入新的元素的时候，需要知道这样的元素在容器里是否已经存在。因此，这和一般的查找算法的时间复杂度是一样的，都是 O(lnN)。另外，插入操作会导致一些小范围的元素的重新排列，平衡二叉树是需要平衡的。大多数集合的实现都会尽量减少插入操作的开销，这靠使用 red-black 树或者其他一些众所周知的算法来实现树的平衡。所以这点额外的开销不会很显著。

遍历集合里的所有元素只要操作指针就可以了，和 list 的情况一样。由于集合里的元素组织成树的形式，虽然这会有一些额外的开销，但是很小。所以集合的总体性能和 list 还是差不多的。

3. 内存使用

如果很关注容器的确切的内存使用量，那关联式容器就不是最好的选择。因为它们的实现使用了平衡二叉树，每一次的插入操作都要求分配一个新的结点，删除操作则会删除结点。幸运的是，重新排列平衡二叉树，使之平衡，并不会导致额外的内存分配。

4. 建议

set 以及 multiset 可能不是日常开发中所需要的容器的类型。但是当程序需要挑出所有元素中重复的元素时，使用集合是不错的办法。根据特定的情况，有时候 map（下一部分会讲到它）也是不错的办法。一些应用集合的常见的场合如表 8.4 所示。

- 保存一系列标准化的向量，因此不需要多次处理同样的问题。
- 保存一系列从硬盘中读取的对象。如果这些对象以时间顺序被读取（从旧到新的顺序），这样新的对象就会覆盖掉旧的对象。把这些数据放到集合里，当我们要读取的时候，可以自动检查是否已经存在，也就不会覆盖掉旧的对象了。

表 8.4 set 和 multiset 总结表

插入/删除元素	O(ln N)
查找元素	O(ln N)
内存分配	每一次的插入和删除操作
遍历性能	比 list 稍微慢一点
是否支持连续的内存访问	否
iterator 有效性验证	从来不做
内存开销	8~12 字节的头部信息，8~12 字节每结点

8.4.2 map 和 multimap

map 是另外一种关联式的容器，因此它的主要用途是提供快速的查找。它和集合很相似，但是使用了新的方法。对于集合的情况，元素本身就是键，可以用来排序和查找操作。对于 map 的情况来说，有两部分数据：元素和键，键用于在 map 中查找元素。

其实可以理解 map 就是一个数组，但是不使用整数作为索引，可以使用任何形式的数据类型作为引用元素的索引。为了方便，map 甚至也可以定制运算符[]。如果这样的话，可以使用和数组一样的语法来访问 map 里的元素了。一个具体的声明可能是这样的：

```
map<Key , Data , Compare , Alloc >
```

这里 Key 是用来映射 map 的键的数据类型；Data 是元素的真实的数据类型；Compare 是一个函数，该函数用来比较元素以及插入元素时的排序。

map 和 multimap 的差异也很容易猜出来，同 set 和 multiset 的关系一样：map 里的键值是惟一的，multimap 却可以有重复的键值。注意区别仅在于键上，与具体的元素没有关系。当然在一个一般的 map 里，可以有相同的元素，只要这些元素有着不同的键值就没有问题。

使用 map 的程序很多，但是有一种常见的类型是在一些任意的数字和一些数据中做快速查找。举个例子来说，考虑一下，游戏世界中的每一个实体都有一个惟一的 ID，这些 ID 都是 16 位的数字，所以数字的范围是 0 到 65536。我们需要一种方法，能够使得我们可以从这个惟一的 ID 来得到这个 ID 对应的实体。

一个可能的办法就是遍历所有的实体，检查每一个实体的 ID，直到我们找到了需要的那个，或者所有元素都找完了就停止。如果游戏世界里的实体比较多，而且这样的查找在每一帧都会调用很多次，那开销就比较大了。

另外一个与之不同的办法是创建一个数组（或者一个 vector），该数组（或者 vector）有 65536 个指针。每当游戏实体创建的时候，就以该实体的 ID 为下标，以指向该实体的指针作为这个数组元素的值。从该 ID 得到实体很容易，因为只要从数组里找出那个实体就可以了。问题是数组太费内存，如果指针是 32 位的，数组本身会占用 256KB，这还不算游戏

世界里的那些具体的实体所占的内存。这看起来已经很浪费内存了，事情其实更糟。也许256KB 算不了什么，但是游戏实体的 ID 不一定限于 16 位，很有可能是 32 位或者更多。一个数组要想包容这么多元素，需要的内存可以高达 16GB。这个数字可比 256KB 大的可怕。所以很明显，需要其他的办法。

map 就是最好的办法。可以创建一个 map，它以整数作为键，以指向游戏实体的指针作为自己的元素。为了得到实体的指针，可以用和数组一样的语法，而且这样的操作相对来说，效率也很高。更为重要的是，游戏的实体有多少，需要的内存就是多少。

map 和 multimap 都是排序的关联式容器，可以以这种顺序遍历 map 里所有的元素。像集合和多集合的例子一样，元素的顺序并不是元素当初插入的顺序，这是因为 Compare 函数起作用的结果。

1. 典型实现

map 和 multimap 的实现和集合一样，使用了平衡二叉树。惟一的区别在于，用来访问和排序元素的键是另外一个不同的对象，而不是元素本身。

2. 性能

map 和 multimap 的性能与集合的性能几乎一样，除了一点有所不同：所有的比较依据的是元素的键，而不是元素本身。如果有着很复杂的元素（比如它占用内存很大），但同时有着很简单的键，那使用 map 可以得到很好的性能。从另一方面来讲，如果键的类型很复杂，比如是 string 或者其他复杂的对象，那两两比较任意两个元素将很慢，这也会导致整个搜索慢下来。

还需要为 map 的运算符[]多啰嗦几句。表面上来看，它是一个很方便的访问 map 里的元素的很好的方法，就像 vector 或者数组那样。必须了解的是，如果选用了 map，那运算符[]并没有像 vector 那样快。即使它给我们这样一种错觉，即搜索的时间复杂度为 O(1)，实际上它的时间复杂度为 O(ln N)。

另外一个有意思的事是关于运算符[]的，当要访问的元素压根就不存在的时候，会发生什么呢？如果我们写的元素不存在，那新的元素就会加到 map 里，并且为它设置一个键。

```
//  A new entry is added to the map as we would expect
map<int, string >playerName;
playerName [0] = game.GetLocalPlayerName();
```

然而，如果试图读取一个不存在的键的对应的元素的时候，会发生什么呢？非常让人感到意外的是，这样的结果是会为 map 增加一个新的元素，并且其键值就是刚才要引用的键。

```
// Try to get the name of player 0, even though there is
// no such player. A new player gets added!
map < int, string > playerName;
```

```
const string & playerName = playerName[0] ;
// Now playerName[0] == ''
```

因此，如果仅仅是想找出某一个元素是否已经存在于 map 中，使用 find 函数就可以了。如果元素确实存在于 map 里，该函数能返回一个指向这个元素的 iterator；如果没有找到就返回.end()。

最后，如果你想往 map 里加一个元素，使用 insert()函数可以获得比运算符[]稍微高一点点的效率。这是因为使用运算符[]，在真正插入到 map 之前，需要创建几个临时的备份来表示“键-元素对”，然后做复制操作。如果直接调用 insert 的话，虽然也要创建这样的“键-元素对”的备份，但是只需要做为数不多的复制操作。然而，当需要修改某个元素的内容的时候，使用运算符[]比使用 insert()函数（使用同样的键值来覆盖已有的这个元素）效率稍微高一点。

3. 内存使用

map 使用了和集合一样的内存使用方式，所以关于内存分配方面和集合的一样，不再赘述。

4. 建议

当需要以句柄、一个惟一的 ID 或者一个特定的字符串来引用一个对象的时候，通常可以把 map 用作一个可以快速查找的字典。map 也可以认为是一种数组，只不过它的下标不是整数，而是其他比较的复杂数据类型，比如字符串或者其他比较复杂的对象。

map 是非常有用的概念，哈希 map（将在下一小节讲述）可以提供同样的功能，但是有着更好的性能。当要在程序里处理大量的元素，除非使用的 STL 没有实现哈希 map，或者内存很吃紧，否则应该考虑使用它。有关 map 的总结如表 8.5 所示，使用 map 的程序一般有以下几种。

- 维护一个 ID 字典，可以通过 ID 来引用对应的游戏实体。这可以避免使用指向实体的指针，因为这些实体可能随时会消失。
- 把字符串转换成整数，例如，把玩家的名字转换成玩家 ID。

表 8.5 map 和 multimap 总结表

插入/删除元素	O(ln N)
查找元素	O(ln N)
内存分配	每一次的插入和删除操作
遍历性能	比 list 稍微慢一点
是否支持连续的内存访问	否
iterator 有效性验证	从来不做
内存开销	8~12 字节的头部信息，8~12 字节每结点

8.4.3　哈希类容器

有一种关联式的容器，没有成为一种标准，但是在少数一些 STL 的实现中，可以使用。它不是一个容器，而是一个家族：哈希的关联式容器。

哈希容器家族包括哈希 set、哈希 multiset、哈希 map 以及哈希 multimap。你也能猜得到，这些容器和 set、multi set、map 以及 multimap 非常相似。实在是太像了，从用户的角度来看，几乎没有什么区别。惟一的差别在于，它们的实现方式、它们的性能以及使用它们所带来的后果。

到目前为止，看到的所有的关联式容器的实现都是某种形式的平衡二叉树，这种数据的组织方式能以很快的速度访问容器的任何元素，这也正是关联式容器的主要目的。这里“很快的速度”指时间复杂度为 O(ln N)，但是有时候可以更快一点。

哈希容器有着和它们对应的容器一样的规则，但是实现的方式不同：它们使用了哈希表。这意味着，只要使用了合适的键和合适的哈希表的大小，查找元素的时间复杂度为 O(1)。不幸的是，如果数据没有正确的设置好，那性能也会很糟糕，时间复杂度会达到 O(N)。假如出现这种情况，还不如平衡二叉树来得快呢。

对于大多数情形来说，哈希容器的性能会很接近 O(1)。这有点像快速排序，理论上相当慢，但是对于大多数随机的序列来说，这通常是最快的排序算法。hash_map 的声明的模板看起来比一般的 map 要复杂一点：

```
hash_map<Key, Data, HashFcn, EqualKey, Alloc >
```

Key 和 Data 的说明和 map 里的一样，容器里要包含的键和元素的数据类型。HashFcn 是一个函数，它以一个键的引用作为参数，返回的是一个哈希键，类型为 size_t。正是这个函数真正把数据哈希化的。容器就是以元素的哈希键为基准来把新插入的元素放到正确的位置上的。EqualKey 是另外一个参数，该参数与 map 的不同。与 map 不同的是，哈希表不会关注两个键哪个大、哪个小。这个函数要做的，仅仅是关注它们的键值是否相等。默认的情况是，hash_map 使用运算符= =来比较两个键值是否相等。

hash_set 以及 hash_multiset 声明的模板是一样的，只是它没有 Data 这个参数，因为参数 Key 同时作为键和数据。

1. 典型实现

哈希容器的实现采用传统的哈希表。有很多“桶子”（buckets，通常是在内存里连续分配的，和 vector 一样），每一个桶子都保存一个指针，该指针指向一个链接表，这个链接表里的所有元素属于这个桶子。

为容器增加一个新的元素，就是用哈希函数求出键，通过该键把元素指向正确的桶子，并把元素插入到桶子的链接表里。元素在链接表里的真实顺序并不重要。

查找元素遵循着相似的过程，但不是增加一个元素。为了寻找想要的元素，需要找到

元素所在的桶，一旦完成了这一步，也就遍历了整个链接表。理想的情况是，那个链接表应该尽量地短，那个链接表仅仅有一个元素。图 8.5 展示了哈希类容器的一个典型实现。

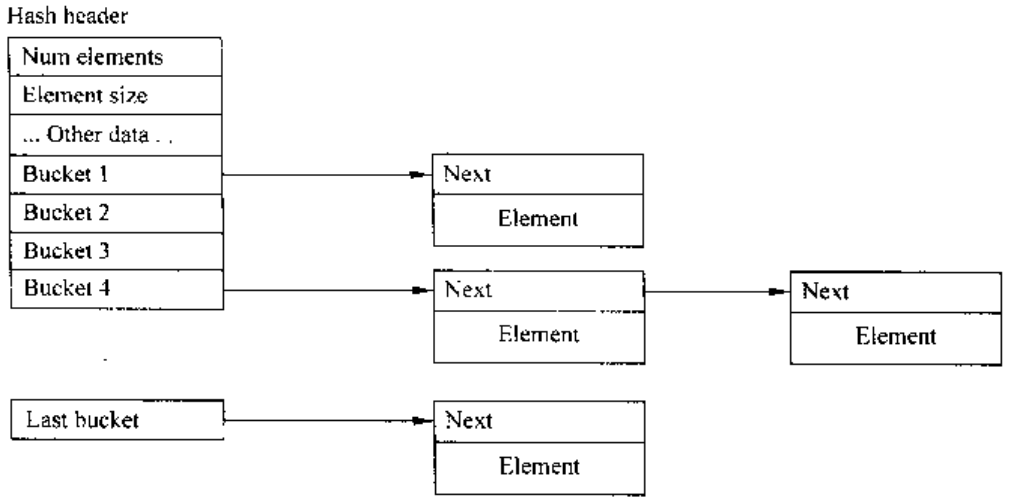


图 8.5　哈希类容器的实现

2. 性能

在关于哈希类容器的简单描述中已经探讨了它们的性能。毕竟，性能是把它们和其他关联式容器相区别的主要因素。正如所看到的那样，在理想的情况下，在哈希类容器里查找元素非常快，时间复杂度为 O(1)。然而，和其他的哈希表一样，事情也有可能变坏。哈希函数为插入的元素新建一个内部使用的哈希键，容器的整体性能完全取决于这样的函数性能究竟怎么样。一个好的哈希函数能够生成分布较好的键，能够避免在一个桶里有多个元素。

可是，一些很特别的键是有可能映射到很少的几个哈希桶里的，这会导致大多数的操作性能下降，时间复杂度可能降到 O(N)。幸运的是，大多数情况下，默认的哈希函数提供的模板函数已经足够好用了，尤其是当哈希键很简单，比如为整数或者字符串的时候。

3. 内存使用

哈希类容器的内存使用情况又怎样呢？不奇怪，哈希类的容器会很频繁的创建和释放动态内存。元素还是以结点的形式存在，这一点比较像排序的关联式容器。因此，插入一个新元素会为之分配新的内存，而删除一个元素会释放内存。

除了结点本身之外，哈希类容器需要为所有的桶子分配内存。桶子本身应该很小。但是要记住，大多数情况是一个元素放一个桶子里。偶尔有时候，也有可能多个元素放一个桶子里。如果哈希表里的元素较小，那桶子会使得总的内存消耗增大到两倍或者三倍。对于大的元素，桶子引起的内存开销不会多于几个百分点。

很有意思的是，哈希类容器有一些函数可以直接处理桶子。bucket_count()函数返回的是目前分配了多少个桶，resize()函数和 vector 的函数相似，但是它设置的是桶子的个数而不是元素的个数。因此，假如知道了哈希表的最少元素的个数，当哈希表不断膨胀的时候，

应当使用 resize()函数调整元素个数到正好，以避免额外的内存分配和桶子的复制操作。

4. 建议

如果一个哈希类容器包含的元素非常多，哈希键又设计的比较合适的话，那它的性能要比排序的关联式容器好的多。哈希类容器的惟一开销是它保存元素不讲究什么顺序，使用的内存多一点。如果想得到最佳的性能，应该适应这些限制，并设计出很好的哈希键。那么，哈希类容器会是很好的选择。

如果性能不是第一位的，那使用排序的关联式容器会少一些风险。使用它，就不用担心乱七八糟分布的数据可能造成的运行期性能下降；而且关联式容器是标准 STL 的一部分，所以也不要担心移植到其他平台会遇到什么问题。

由于游戏里性能常常是最关键的，所以哈希类的容器要比一般的排序关联式容器用途广得多（如表 8.6 所示）。哈希类的关联式容器适用于以下一些情形。

- 维护一个惟一的 ID 的字典，每一个 ID 对应一个游戏实体。游戏可以避免使用指针来引用一个随时可能消失的游戏实体。例子和在 map 中讲述的一样，但由于往往用于每一帧都执行很多次的查询，所以使用哈希类的容器还是值得的，这样能够获得最好的性能。
- 通过类的惟一 ID 来快速的查找出类的信息。这可以用来实现一个效率很高的 RTTI 系统（参见第 12 章运行期类型信息）。

表 8.6　哈希类的关联式容器总结表

插入/删除元素	O(1)－O(N)
查找元素	O(1)－O(N)
内存分配	每一次的插入和删除操作
遍历性能	比 list 稍微慢一点
是否支持连续的内存访问	否
iterator 有效性验证	从来不做
内存开销	取决于具体的实现；对于一个分布良好的哈希表，每一个元素使用 8~16 字节

8.5　容器适配器

到目前为止，已经了解了 STL 所提供的全部容器类型。有一些是期望的，比如 vector（其实也就是一个可以动态改变大小的数组而已）、list 或者 set。也有一些不是很知名的容器，比如 deque 和 multimap。但是其他的一些非常基本的数据结构情况怎么样呢？STL 提供了栈和一般的队列了吗？

考虑一下要支持栈这样的容器必须要做的的操作。也就是在栈的顶部压入一个元素，

同时也是在顶部弹出元素。任何序列式容器都支持这样的操作，所以栈可以看作是一个给定了限定条件的某种容器。

如果 STL 的实现提供了栈这个容器，想想它是怎样实现的。也许是用 vector 实现的，那么当元素超过一定数量的时候，需要重新分配内存吗？或者使用 list，这样每一个元素就是一个结点；或者使用 deque，使用以内存页为基准的内存分配。所以这些可能的实现都是有效的，我们可能在不同的程序里需要不同版本的栈。

因此，STL 不是仅仅给一个栈的实现，它提供的是容器适配器。容器适配器可以为一个已经存在的容器提供有着限定条件的新的接口，而不用为已有容器增加任何新的功能。

为什么提供容器适配器来限定可以直接操作的容器呢？看起来好像根本没有必要使用容器适配器。如果想把 vector 当做栈来使用，直接写代码就可以了。实际上，确实是这样。然而使用容器适配器，有两个主要的原因。

- 声明一个数据结构为栈，每一个阅读代码的人就会很清楚，我们会对它进行什么操作，它会遵循什么样的规则，这要比使用 vector 能提供更多的信息。
- 除了可以让其他的程序员了解要对该数据结构进行的一些操作，还可以告诉编译器，我们不会使用一个根本不属于栈的接口。如果使用的是 vector，那就很容易看到栈的第二个元素，这会破坏为该数据结构（指的是栈）所设定的规则。

8.5.1 栈

栈是最简单的容器适配器了，就像刚才看到的那样，对栈能做的操作主要有在栈的顶部增加和删除元素，还能查看栈顶元素的值。

由于栈比其他的序列式容器限定要严格，所以不能直接在容器上直接操作。很明显，栈的性能、内存特性和它使用的容器的一样。但是默认情况下，栈是用 deque 来实现的。下面的代码实现了游戏中的事件系统，最后一个事件必须立即处理。

```
// Create a stack using a deque for game events
stack <GameEvent> eventStack;
eventStack.push( currentEvent );
// ...
// Resolve all events in the stack
while ( !eventStack.empty() ) {
   GameEvent & event = eventStack.top();
event.Process();
eventStack.pop();
}
```

如果觉得 deque 实现的栈不够好，可以使用 vector，惟一的改变就是定义 eventStack 这个变量而已。

```
// Create a stack using a stack
stack < vector<GameEvent> >eventStack;
```

8.5.2　队列

队列是另外一个使用很广的数据结构，STL 也不应该缺少它。和栈的情况一样，队列也是某个容器的限定版本。队列允许的操作仅是在尾部添加元素，在头部删除元素。我们不能在别的位置添加和删除元素，甚至不能访问队列中的其他元素。

默认情况是采用 deque 实现队列，但是也可以采用 list 实现队列。有点奇怪，即使明白在头部删除元素，其性能不会太好，也不能采用 vector 来实现队列。这是因为 vector 没有 push_front()函数，而这恰恰是队列这个容器适配器所需要的。

```
// By default the queue is using a deque
queue < Message > messages;
messages.push( getNetworkMessage() );
// ...
// Process all messages in the order they were received
while( !messages.empty() ) {
   const Message & msg = messages.front();
   process( msg );
   messages.pop();
}
```

为了使用同样的代码用 list 实现队列，可以用 list 容器声明队列。

```
// Now the queue is using a list
queue < list <message>>messages;
```

8.5.3　优先队列

优先队列和一般的队列很像：只能在队列的尾部增加元素，也只能在队列的头部删除元素。区别在于，要在头部删除的那个元素不是最先插入的那个元素，而是具有最高优先权的那个元素。

和其他的容器适配器不太一样，优先队列适配器多加了一点点功能。当元素要插入到容器里的时候，需要根据一个优先权评估函数来排序。这事很容易就可以做到，实际上这不能算作加了新东西，它只不过是把一些高级功能包装到适配器里而已。

默认的情况，优先队列使用 vector 作为它的底层表述，也可以使用 deque。但是不能使用 list，因为为了获取很高的效率，优先队列需要随机访问元素来完成元素的插入操作，还要为之排序，而 list 并不支持随机访问元素的能力。

元素之间比较的结果决定元素的优先权。同样也是默认的情况，这是靠使用 less<Type> 来实现的。如果想用不同的函数来实现自己的优先权比较，或者元素没有定义运算符“<”，那就必须提供比较函数了。

以下的代码片段展示了一个优先队列，它的元素的优先权是依靠距离确定的。游戏中应用优先队列的场合有以下几种：听到的声音有远近之分；光线的远近对游戏对象的影响也不一样；或者某游戏实体的 AI 依据距离的不同来确定下一步怎么做等。

```
struct EntityPos {
   EntityPos ( uint nUID, const Point3d & pos);
   Uint nUID;
   Point3d pos;
};

template <typename T>
class CloserToCamera
{
   public:
        bool operator() (const T & p1, const T & p2) const;
};

bool CloserToCamera <typename T> ::operator() ( const T & p1,
                                                const T & p2) const
{
   const Point3d & campos = GetCurrentCamera.GetPosition();
   Vector3d camToP1 = p1 - campos;
Vector3d camToP2 = p2 - campos;
return ( camToP1.length() < camToP2.length() );
}

priority_queue <EntityPos, CloserToCamera<EntityPos> >entities ;
//  Fill all the entities we might want to consider
entities.push( EntityPos(nUID, pos) );
// ...
//  Now process entities in order of distance to the camera until
//  We run out of time, or maybe a fixed number of them
while( WeShouldContinue() ) {
   Uint nUID = * ( entities.top() ).nUID;
entities.pop();
//  Do something with that entity
}
```

8.6　结　论

在本章的讲述内容里，了解到几乎没有什么理由不在开发当中使用 STL 的，无论是开发游戏还是开发工具软件。可是，当使用不同容器的时候，需要搞清楚它们的性能以及内存使用方面需要注意的问题。事实上有一些容器完全不适合某些特定的任务，有时候需要放弃 STL 不用，而使用更为底层的代码。容器有两种主要的类型：序列式容器和关联式容器。

序列式容器：序列式容器允许使用者知道元素保存的顺序。可以在容器的任意位置添加和删除元素。但是不同的容器其性能差异明显。主要有以下容器：

- vector：vector 就像一个可变大小的数组，元素顺序地保存在一个内存块中。当需要更多的内存的时候（比如新插入的元素比较多），该内存块会改变大小以适应更多的元素。在尾部以外的任何地方执行添加或者删除元素相对来说会比较慢。
- deque：deque 和 vector 比较像，但是其插入和删除元素的操作，无论在头部还是尾部，效率都很高。deque 使用了一些内存页，元素在每一个内存页里是顺序的。
- list：list 可以使插入和删除操作无论在什么位置都很快，但是付出的代价是，需要额外为每一个元素支付一定数量的内存，从而使得总内存使用量变大。元素在内存中不是顺序保存的，这使得遍历一个较大的 list 的时候，数据缓存的一致性变差。

关联式容器：关联式容器提供了快速查找出特定元素的能力。关联式容器以自己的顺序维护元素，而不是使用者最初插入的顺序。

关联式容器有两个分支，排序的关联式容器和哈希的关联式容器。关联式容器的四种类型都有这两种分支。

排序的关联式容器的典型实现为平衡二叉树。元素以容器维护的顺序保存在平衡二叉树的结点上。访问元素的时间复杂度为 O（ln N）。

哈希的关联式容器的典型实现为哈希表。元素的保存没有任何顺序。访问其中的元素的时间复杂度最好的情况为 O(1)，最糟糕的情况为 O(N)。

关于关联式容器的总结如表 8.7 所示。

表 8.7　关联式容器总结表

容　器	是否可以拥有同样元素的多个实例	键是否是元素本身	是否实现为哈希表
set	否	是	否
multiset	是	是	否
map	否	否	否
multimap	是	否	否
hash_set	否	是	是

续表

容　器	是否可以拥有同样元素的多个实例	键是否是元素本身	是否实现为哈希表
hash_multiset	是	是	是
hash_map	否	否	是
hash_multimap	是	否	是

容器适配器：除了提供各种不同的容器之外，STL 还提供了容器适配器。容器适配器在现存容器之上提供了新的接口，用来实现其他一些常用的数据结构。STL 里提供的三种容器适配器是栈、队列以及优先队列。

8.7 阅读建议

这里有一些介绍 STL 的书，这些书里面也有讲 STL 比较深入的，但是它们都是从很简单的地方开始说起的。

Musser, David R.,Saini, Atul, STL Tutorial and Reference Guide, Addison-Wesley, 1996.

Austern, Matthew H., Generic Programming and the SiL, Addison-Wesley, 1999.

Josuttis, Nicolai M., The C++ Standard Library, Addison-Wesley, 1999.

下面这本书是为数不多的关于 STL 的书籍中深入浅出的讲解如何在程序里更有效地使用 STL 的。在阅读了本章之后，如果急于了解更多的细节，强烈建议看看 Meyers 的这本书。

Meyers, Scott, Effective STL, Scott Meyers, Addison-Wesley, 2001.

至于每天工作的手头参考，可以查阅在线参考材料：

SGI 的 STL 参考，包含哈希以及其他一些不标准的容器，http://www.sgi.com/tech/stl/。

第 9 章　STL 算法及高级主题

本章将讲述：

- 算符（函数对象）
- 算法
- 字符串
- 分配算符（高级话题）
- 当 STL 不满足需要时（高级话题）

第 8 章探讨了 STL 的基础——容器。即使用户在程序中只使用容器，学习 STL 也是值得的。通过学习它，可以了解并掌握到很多种经过深度调试和优化的数据结构。

然而，STL 不止是容器，它还有更多功能。本章将包括两个主要方面，一方面介绍可以用在容器上的不同类型的已经实现的算法，例如 string 字符串类，另外还要对编写自定义的分配算符、跟踪 STL 内存等游戏开发更有帮助的高级主题进行论述。

9.1　算符（函数对象）

在深入介绍算法之前，首先需要了解函数对象（或者算符）。实际上，在第 8 章已经介绍函数算符了。但是在谈到算法的时候，算符就更为密切相关。

9.1.1　函数指针

这里首先给出一个例子，这个例子的情形与第 8 章使用函数指针的情形类似。假设现在有一个容器，需要决定如何对容器中的元素进行排序。要想非常灵活地为这些元素排序，很显然，就需要编写某种函数，通过函数来得到两个元素，并判定其中哪个元素“较大”。那么，要是把这些函数传递到容器代码中，使用什么方式最好呢？

传统的方法是使用函数指针实现的。下面给出的代码讨论了这一方法，在这种方法里，首先创建一个比较函数，对两个数字进行绝对值比较。接着创建一个可以传递到其他函数的函数指针。

```
bool  LessAbsoluteValue(float a, float b)
{
```

```
    return (fabs(a)<fabs(b));
}
bool (*mycomparison)(float, float);
mycomparsion = &LessAbsoluteValue;
//Now we can pass mycomparison to any function that takes
//function pointers of that type
```

函数指针看起来可能有点混乱，显得不够美观。尤其是开始把大量的圆括号堆积到一起，并把它们作为函数参数使用的时候，这种现象就更突出。例如，这里给出一个函数声明，功能是实现浮点数矢量的排序，给定任一比较函数，将指向比较函数的指针作为排序函数的参数使用。

```
void sort (bool(*cmpfunc)(float, float),vector<float>);
```

通常遇到看起来不够清晰的代码，并且最重要的是很难快速分析出函数的参数。为了程序代码的可读性，可以对函数指针类型使用 typedef 定义。

```
typedef  bool (*ComparisonFunct)(float,float);
ComparisonFunct mycomparison=&LessAbsoluteValue;
void sort (ComparisonFunct,vector<int>);
```

这段代码的易读性和前面的那段相比就显得好一些。讨论到现在，一直假定需要排序的元素类型是已知的。但是，如果想要排序函数对任意类型的元素都可以进行的话，就需要使用模板了。模板函数指针的语法更不易读，并且把情况搞得更糟的是，C++不允许使用模板 typedef。因此，通常采用的方式是把 typedef 封装到模板类中，这样就可以得到需要的功能，但是付出的代价是编写了更多的代码，并且同时增加了代码的复杂性。

9.1.2 算符

毫无疑问，STL 中采用了另外的办法。在 STL 中，把函数传递给容器和算法是一项广泛使用的技术。因此，使函数的传递尽量简单明了是很重要的事情。

在 STL 中使用的方法是利用函数对象，也可称为算符来实现的。算符就是指可以被称为函数的任何东西。比较典型的算符就是实现了 operatro()方法的对象，但是函数指针甚至函数本身也可以认为是一个算符，并且可以在 STL 中使用。

```
class EvenNumbersFirst{
public:
      bool operator() (int a, int b) const{
            return ( fabs(a) < fabs(b) ); }
};
```

为了将同一函数适用于不同的数据类型，可以采用与模板一样的方法对函数进行处理。在下面这个例子里，有效的数据类型是函数 fabs 可以接受的数据类型，因此仍要受到限制。

```
template <typename T>
class EvenNumbersFirst{
public:
   bool operator() (T a, T b) const{
        return( fabs(a) < fabs(b) ); }
};
```

现在在 STL 自身的 sort 函数中使用这种新的、模板化的函数就显得很简单了。在下面的例子里，限定比较函数对两个浮点数进行比较。

```
sort(sequence.begin(),sequence.end(),
     EvenNumbersFirst<float>);
```

除了可以把模板函数作为参数传递这一优点之外，算符还有其他几个优点。首先就是方便性。其次，一个算符就是一个对象，因此算符本身可以包含成员变量或者其他辅助函数。如果算符内部封装的是复杂函数，那么这些函数的相关数据或者辅助的功能都会作为算符的组成部分。

例如，假定需要对在所有计算中使用的引用次数进行管理。这个数字可以作为一个成员变量在需要的时候使用。如果同时使用了同一算符的几个实例，那么每个不同的实例可能会拥有自己的引用数。而实现相同的功能，如果使用函数指针的话，通常需要一个全局静态变量或者栈，并且该函数还要能够感知到其他类似函数的存在，这就会造成程序不必要的复杂性。

算符的另外一个优点表现在性能上。当使用函数指针时，编译器必须等到运行时刻，否则无法灵活地控制函数调用。而这些类型的函数通常是一些调用频繁的小函数，这样就可能导致性能开销很大。

而当使用带有模板类或者模板函数的算符时，就可以通知编译器需要调用什么特定的函数。编译器就能够优化代码，并且通常会把模板代码内嵌到函数中。对于小型的、调用频率较高的函数，这样的方法可以显著提高执行性能。

9.1.3 算符适配器

经常会遇到的一种情况是想传递一个特定类的成员函数。不可以把指向函数的指针进行传递（成员函数具有“隐藏”参数，这个参数是一个指向函数作用对象的指针）。因此，很显然，惟一的方法就是创建一个类来封装算符，利用这个类来调用需要的函数。这种方

法可以实现以上功能，但是却显得复杂了许多。可喜的是 STL 提供了算符适配器。

算符适配器就是模板，这些模板允许将已有的成员函数作为算符，并可以在任何可以使用算符的场合使用这些函数。很显然，与调用其他算符一样，调用这些函数也需要具备与调用代码希望相一致的参数个数和类型。算符适配器表现在三个方面：

- mem_fun：利用指针对成员函数进行操作
- mem_fun_ref：利用对象或者引用对成员函数进行操作
- ptr_fun：利用函数指针对全局函数进行操作

现在给出一个例子，假定再次对元素进行排序，但是这次处理的元素不是整数，而是对象。更确切地说，这些对象是场景节点，我们想对这些场景节点进行排序，以便使得渲染状态改变最小，并且保证尽快对节点进行渲染。如果有一个全局函数对两个场景节点进行比较，然后判断哪个节点需要先渲染，要完成这样的功能，就需要把函数指针传递到排序函数。但是如果排序函数是场景节点自身的一个成员函数时，该怎么办呢？这里就需要使用到算符适配器。

```
vector<SceneNode *> nodes;
//…
sort(nodes.begin(), nodes.end(),
        mem_fun( &SceneNode::RenderFirst) );
```

注意在这里使用了 mem_fun，原因是容器里包含有指向对象的指针。如果实际中直接在向量里保存了 SceneNode，那么就要相应地使用 mem_fun_ref 了。

严格地讲，多数情况下不是特别需要 ptr_fun。算符适配器除了把函数绑定到算符之外，还要通过定义算法中使用的一些 typedef，在场景背后做一些额外的工作。如果想直接把函数指针传递到这些算法，那么将会遇到语法错误，这些语法错误可以通过使用 ptr_fun 适配器修正。

9.2 算　法

算法在 STL 中占了很大分量的内容，算法是可以对容器执行通用操作的模板函数。因为这些函数本身是模板化的，所以也适用于任意数据类型。另外，因为这些函数作用在已存在的容器上，所以函数知道如何进行转化和调整。

依据这些算法的行为，可以总体划分为四类。虽然这种划分有点武断，但是与罗列出几百个未分类的函数相比，还是要好出许多。此外，要是用户知道自己想要找什么内容的话，基于这些分类就可以很容易地找到需要的正确算法。

多数算法带有来自同一容器的两个 Iterator 作为参数。第一个 Iterator 指向应用当前算法的序列中的第一个元素，第二个 Iterator（也是 STL 的习惯）指向应用当前算法的序列中的最后一个元素的下一个元素。

有几个算法要求有一个值作为参数进行传递，这个值决定对容器进行的操作类型，例如查询算法（search）、替换算法（replace）以及取元素数量的算法（count）等。这些算法中，多数算法是把一个谓词函数作为变量使用，而不是使用一个数值。所谓谓词就是返回类型为真或假的一个函数，而且通常以算符形式实现。算法函数的谓词版本提供了更多的灵活性，操作对象可以是谓词限定的全体元素，而不只是单一的匹配元素。带有谓词算法的版本通常是在原来版本名字后面附加_if形成的。因此在使用过程中可以非常清晰地知道使用的是哪个函数，而不是依赖多态函数调用。

这一节将列出在游戏中最常用的STL算法，并通过例子给出算法应用的特定情形。为了对所有算法有一个全面描述和了解，可以参考本章最后给出的关于STL方面的参考书目。

9.2.1 无改变算法

无改变算法，正如算法名字所言，可以对容器进行操作但是丝毫不改变容器内容。例如，查找一个元素或者计算元素的个数就属于无改变算法。

1. Find

可能最常用的无改变算法就是查找算法find。该算法通过在序列中（序列由作为参数传递的两个Iterator决定）的所有元素中迭代，来查找特定的元素。

查找算法具有线性时间特性O(N)，因为该算法不对它访问的容器做任何特定假定，因此需要遍历每个元素，直到找到希望查找的元素。关联式容器在查找特定元素方面提供了更好的性能。这种情况下，需要使用成员函数find而不是单独应用查找算法。关联式容器由于自己独特的组织方式，因此其效率较高，时间复杂度为O(lnN)。

查找算法find用于在未排序的容器元素序列中查找元素。为了避免不必要的性能损失，只是偶尔使用，或者只在元素数量较少的情况下使用。例如，可以搜索游戏玩家列表或者搜索本地游戏玩家已经建立的不同人物，但是尽量要避免使用这一算法在游戏中搜索指定的游戏实体。

```
list<string> PlayerNames;
//...
//Make sure the player name is not taken for this session
if( find(PlayerNames.begin() , PlayerNames.end() ,
wantedName) == PlayerNames.end() ){
    //Name was taken, do something
}
```

这里给出了查找算法的一个谓词版本find_if。该算法可以在谓词判断为“真”的序列中搜索一个元素。例如，下面给出的代码在游戏实体ID中寻找那些数值小于10 000的实

体 ID（可能是因为低于该范围的 ID 是为其他目的而预留的）。

```
class LessThan10K{
public:
   bool operator(int x) const { return x < 10000; }
};
vector<int> GameUIDs;
//…
vector<int>::iterator it;
it = find( GameUIDs.begin() , GameUIDs.end() , LessThan10K() );
if ( it != GameUIDs.end() ){
   //We found at least one
}
```

2. for_each

该算法对序列中的每个元素执行特定的函数。正如希望的一样，序列的范围是通过两个 iterator 指定的，执行的特定函数是通过算符传递的。

拿游戏编程里最常用的情形作为例子来说明。我们的任务是在特定的框架下对所有的游戏实体调用 update 函数。这样就赋予游戏实体执行 AI，更新位置，开始新的动画等动作的机会。实现这一功能时，可以想到的最自然的方法就是建立一个 for 循环：

```
   //EntityContainer is a typedef for the specific container of the game
entities.
   EntityContainer::iterator it;
   for (it = entities.begin(); it!=entities.end(); ++it){
      GameEntity & entity = *it;
      Entity.Update();
   }
```

现在，可以用 for_each 算法完成相同的功能：

```
   void Update (GameEntity & entity ) {
      entity.Update();
   }
   for_each (entities.begin() , entities.end() , Update);
```

因为没有使用循环，所以只是结构上显得清晰，但是却要求构建一个全新的函数。正如前面讨论的，不能直接把成员函数作为算符，所以需要新的 Update 函数。所幸的是，可以利用 mem_fun_ref 来实现。新的代码如下所示：

```
for_each( entities.begin(),entities.end() , mem_fun_ref (&Game Entity:
:UpDate) );
```

这样看起来好多了，但是这种方式真的强于使用简单的for循环吗？坦白讲，不见得。两种方式基本相当。对多数人而言，使用 for 循环方式在易理解方面具有优势，但是使用for_each 方式尽管显得简明，却有点容易出错。具体选择哪种方式就在更大程度上取决于个人的喜好了。

然而，最让游戏开发者感兴趣的就是 for_each 算法的性能优势。for_each 算法在应用到容器时，有可能利用容器的潜在实现，与普通循环相比，可以更快速地执行查询。但是这在很大程度上要依赖于容器以及 STL 的实现；如果对性能感兴趣的话，那么就需要对目标平台多做一些工作。但是有一点可以确定，那就是 for_each 算法绝不会比自己手工编写的任何循环代码执行效率低。

3. count

如果只想知道容器里容纳着多少个元素，那么应该使用的成员函数是 size()。这一算法不只计算容器中的所有元素，而且可以计算与某一特定值相匹配的所有元素。

假设有一个游戏实体 UID 的 vector，当实体被移动以后，把它们的入口置为零而不是移动它们。因而，我们经常想了解究竟多少入口被置为零。确实，可以使用另外一个 for 循环来实现，但是算法 count 却是最好的工具。

```
int numZeros = count( UIDs.begin() , UIDs.end() , 0);
```

count 算法最有用的版本可能是它的谓词变量版本：count_if。利用它可以计算使谓词判断为真的元素的个数，而不是计算带有某一数值的元素个数。以下的代码使用 count_if 来计算距离游戏者一定距离的实体个数。可以利用这个值来改变音乐节拍或者在屏幕上进行动画提示，提醒游戏玩家有大量的敌人朝他的方向走来。

```
class IsEnemyNearby
{
public :
        bool operator() ( const GameEntity & entity)
        {
            return ( entity.IsEmpty() && ::dist(player.GetPos() , entity.
                                                GetPos() ) < RADIUS);
        }
        static float RADIUS = 100.0f;
};

int nNearbyEnemies = count_if(entities.begin() , entities.end() , IsEnemy
                                                        Nearby() );
```

很显然，对于类似在容器中遍历所有元素这样的函数，我们必须注意。如果游戏中的实体数目很大，那么或许在全部遍历这些元素之前，不是只粗略地依据游戏者一定距离范围这样的条件，而是要对所有实体做一个更快的剔除。

4. 其他的无改变算法

其他的无改变算法包括 adjacent_find、mismatch、equal 以及 search。这几个算法不如前面讨论的几个算法使用频繁，但是，仍然有必要掌握这些算法，以便在需要的时候恰当地使用这些算法。可以参照本章最后的参考书目，更为全面地了解每个函数的详细说明。

9.2.2 有改变算法

有改变算法在容器上发生作用并同时改变容器的内容。这类算法不包括排序算法，排序算法属于 STL 算法中的另外类型。有改变算法可以实现复制数值，倒转元素次序，改变元素的顺序等功能。

1. copy

copy 算法把由两个 iterator 指定的序列内部的所有元素复制到另外一个序列，这两个序列可能处在不同的容器。如果只想把一个容器中的所有元素复制到同类型的另一容器中，最简单的方法就是复制整个容器。

```
vector<int> highScores;
//…
vector<int> newScores = highScores ; // Copies all the scores
```

当只需要复制指定序列中的元素，或者想在不同类型的容器之间复制元素的情况下，复制算法就显得非常方便。

```
//Only copy the first 10 scores (assume there are at least 10)
list<int> newScores;
copy(highScores.begin() , highScores.begin() + 10 , newScores.begin() );
```

可以注意到，copy 算法带有三个 iterator。前两个表明复制元素源序列的起始点和结束点，而第三个参数表明了目标序列的开始点。因为前两个参数已经决定了元素的个数，所以没有必要在这里指定目标序列的结束点。

只要满足一定的条件，copy 算法可以使用在序列重叠的场合。只要目标序列的起始点没有覆盖源序列，那么元素就会按预期的方式逐个实现复制。

如果需要在两个重叠的序列间实现复制，所谓的重叠序列指目标序列的第一个元素是源序列的一部分，那么需要使用变量 copy_backward，该变量会从序列的末尾开始复制，这

样就避免了重叠问题。

2. swap_ranges

该算法类似于 copy 算法，带有与 copy 算法相同的参数，用以确定序列范围，实现的是交换两个序列之间的元素，而不是将一个序列中的元素复制到另一个序列。

3. remove

最后讨论一下 STL 中最有用，同时也是最不直观的算法之一：remove。看到下面给出的代码，或许想到该调用会删除序列中具有给定值的所有元素：

```
remove (first , last , value);
```

可以说，差不多是这样的，但是也不完全是。remove 调用实际上没有删除任何元素，而采用的做法是，把匹配值 value 的所有元素放到序列的末尾，并且返回了一个新的 iterator，即 newlast，这样的结果就是从起始点到 newlast 序列之间的所有元素没有等于给定值的元素存在。这一点非常重要，接下来用一个简单的例子来说明。

```
vector<int> test;
test.push_back(3);
test.push_back(1);
test.push_back(4);
test.push_back(1);
test.push_back(5);
test.push_back(3);
test.push_back(1);
test.push_back(8);

remove(test.begin() , test.end() , 1);
```

上面对 remove 的调用将返回 test.begin()+5，因为算法从序列中移出了 3 个元素，那么现在位于 test.begin()到 test.begin()+4 序列之间的所有元素都确保不等于 1。很显然，序列中现在的前 5 个元素分别是 3、4、5、3 和 8。剩下的 3 个元素怎么样了呢？它们就是删除的 3 个元素吗？标准地说它们是没有被定义，因此可以是任何内容。实际上，后 3 个元素通常是保持了初始序列中的内容；它们没有进行任何改动，但是可以明确地讲，它们不是已经删除的 3 个元素。

因此，remove 不会从容器中移出任何东西，元素的数量仍然是不变的。算法的名字或许有点误导，但是没有更好的词语来表达该算法可以完成的功能。

那么如果确实想把一些元素从容器中删除该怎么办呢？应该调用容器中的 erase 函数，来真正地从容器中把一些元素彻底清除。在上面的例子中，如果为了把容器中的元素缩减为只包含不等于给定值的元素，则可以这么编写代码：

```
vector<int> newEnd = remove(test.begin() , test.end() , 1);
test.erase( newEnd , test.end() );
```

一旦熟悉了这些概念，就可以在编码中把它们放在一行，以下就是最常见到的代码方式：

```
test.erase(remove(test.begin() , test.end() , 1) ,test.end() );
```

读者可能可以想象到，remove 算法有一个谓词变量称为 remove_if，该算法不只匹配一个值，而是由算符对每个元素进行评定。

4. 改变顺序算法

这里有一组有改变算法，它们改变序列中的元素次序。另外，这些算法不是排序算法（可以参考下一节的排序算法）。

- reverse(first, last)：颠倒序列中的所有元素次序，该算法把第一个元素放到最后，第二个元素放到倒数第二，依此类推。
- rotate(first, middle, last)：该算法旋转序列中的所有元素。可以把这一算法看作是包含回绕处理的转换。在序列中的元素会被移动，直到中间元素到达第一位置，并且被移出的元素附加到序列的尾部。
- random_shuffle(first, last)：该算法打乱位于序列内部的所有元素。游戏中这种情况非常显然，不仅纸牌游戏如此，每次都需要事先打乱一系列事件，并确保所有元素都得到使用（与其对立的是每次只随机地挑选其中一个）。

5. 其他有改变算法

这里有很多种有改变算法，它们彼此的用途各不相同。在本章没有讨论的算法包括 transform、replace、fill、generate、unique 和 partition，其中有几个算法稍有差别。如果想详细了解，可以参照 STL 参数书目以获取更为全面的了解。

9.2.3 排序算法

在游戏编程中，排序是一种重要的操作。其发生频率远远超出我们的想象。从游戏积分板上的游戏者排名，一个 AI 单元的潜在威胁排序，到渲染前的最小化状态改变需要网格排序——排序功能都是一项主要任务。在 STL 中，尽管 sort 算法是主要的排序方法，但是还有一系列的算法可以辅助我们进行分类。

在 STL 开发之前，惟一的选择是使用标准 C 程序库中的 qsort 或者使用自己的分类函数。标准 C 中的 qsort 尽管大多情形下非常高效，但是其函数形式不太好看，并且要求带有 void 类型的函数指针，这就存在了类型不安全的语法问题，所以不使用这个函数。

所有分类函数中的核心是 sort 算法。这一算法通过使用运算符 operator<，或者作为参

数传递过来的算符，来实现对序列中的元素进行比较。sort 算法利用 quicksort 进行实现，这样就导致 O(N ln N)的平均排序时间。但是在有些情况下，也会存在 $O(N^2)$时间特性。

因为 sort 算法需要随机访问任一元素，所以只可以使用在提供随机访问（例如 vector、栈，或者带有随机访问属性的自己定制的容器）的容器上。以下的代码片断实现在游戏者的最高积分基础上对游戏玩家指针的 vector 进行排序。

```
class HigherScore{
public:
   bool operator() (const Player & p1, const Player & p2) const{
         return ( p1.GetScore() > p2.GetScore());
   }
};

vector<const Player * > players;
sort( players.begin() , players.end() , HigherScore() );
```

分类算法 sort 的一个最大特征就是它的不稳定性。这意味着具有相同排列顺序的两个元素在进行排序操作以后可能位于相对不同的位置。这一点在一般情况下，也就是大多数的游戏开发的情况下，不是一个问题。但是万一对稳定排序要求比较高，就需要清楚地意识到这一点。

如果需要进行稳定的分类，应该使用 stable_sort 算法。这个算法接口与 sort 算法相同，但是却可以保证排序是稳定的。不像 sort 算法，stable_sort 运行速度要慢一些，时间特性为 $O(N \ln N^2)$，因此该算法只在确实需要进行稳定排序时才使用。

有时有数量非常多的元素，但是关心的是最上面的 X 个元素的排序，这种情况下，就可以使用 partial_sort 算法，该算法将最上面的 X 个元素分类，然后把它们放在序列的前部，而把序列中其余的元素保持未指明的次序。partial_sort 算法的性能是 O(N ln X)。在只需要对顶部的少量元素进行排序时，该算法性能远远优于 sort 算法。

STL 还提供了其他几个排序相关的函数，例如合并排序后的序列，判断序列是否已排序等。

9.2.4 通用数值算法

最后，STL 提供了少量对容器进行操作的数值算法。这些算法在游戏开发中不经常使用，但是总有一天它们可以派上用场，因此有必要清楚如何使用这些算法。

- accumulate(first, last, init)：该算法对给定范围内的所有元素进行求和。求和初始值由 init 设置。默认情况下该算法对元素求和，但是也可以通过传递算符来执行其他操作。

- partial_sum(first, last, output)：用于对一定范围内的元素部分计算求和，并在单独的容器中保存。局部求和定义为：对于每个一输出元素，把所有的元素累加到第 n 个元素形成的数字序列。与 accumulate 一样，也可以提供操作符来替代默认累加操作。
- adjacent_difference(first, last, output)：计算相邻元素对之间的差值，并将结果存储到由 output 指向的序列中。通常可以传递任意的操作符来替代减法操作。
- inner_product(first1, last2, first2,init)：首先，这一算法看起来对游戏开发者非常有用。毕竟，内积与点积是相同的，并且在代码上经常进行这些操作，从镜头移动到对 AI 进行路径标图，到高亮方程。不幸的是，游戏开发中的多数点积是在给定的矢量类上进行，而不是直接针对容器操作，所以该算法不像它看起来那么有用。对于这一算法也可以传递定制的算符来替代默认的加法和乘法。

一些 STL 实现不提供标准之外的额外算法。对于用户来讲，如果跨平台要求不是主要问题（或者是 STL 实现都通过目标平台提供），那么可能想知道 STL 提供了什么扩展，以及如何让它们来适应自己的需要。

9.3 字 符 串

STL 也提供了 C++中经常使用到的一个类：string 类。这一节讨论为什么字符串那么有用，以及如果更有效地使用它。

9.3.1 没有提供字符串

与大多数语言不同，C 和 C++本身没有自己的字符串概念，因而必须使用字符数组及约定来进行字符串处理，例如用空字符表示字符串的结束。遗憾的是，用字符数组进行操作显得非常不方便。

当字符串动态改变的时候往往需要创建足够大的数组来容纳字符串，这样就导致以下问题：多大是足够大？例如，一个函数要得到一个文件的全路径，然后把文件路径复制到一个作为参数进行传递的数组里，这就要求为这个数组分配可以容纳整个字符串的空间。为了实现这个目的，每个操作系统都提供了最大常量，例如 MAX_DIR_PATH，这就给出了文件路径长度的上限。

```
char path[MAX_FILE_PATH];
GetFilePath(filename, path);
```

这种情况经常发生。当得到网页上的内容时，在开始下载以前，无法知道网页究竟有多大。因此所有的各类复杂函数都设计为可应用于固定尺寸的字符数组。

字符数组容易出错，往往把情况变得更为糟糕。控制字符串结束的惟一方法是把最后一个字符设置为null或者简单的设置为零。这就意味着数组分配的字符个数总比实际的字符串长度多出一个。这样就导致在字符串长度和数据尺寸转换时的加1和减1混乱。

那么如果在编码中受到了加1这样的转换困惑，或者只是简单地读取或者写入时超出了数字的结尾会发生什么呢？各种各样的事情都可能发生，这些就取决于平台和自己的幸运程度了。如果幸运的话，只要试图访问超出数组边界的任何内容，就会造成程序的立即崩溃。那么可以立即改正整个错误，则预防了后续的头痛事情。如果不太幸运，程序会像平时一样工作，偶尔可能导致奇怪的程序崩溃或者细微的错误，这是一类不可重复的，不可能跟踪的错误。这些错误就是可怕的缓冲区溢出。如果在偶然写入的时候，操作了数组的边界，但是这段内存位置正好被程序的其他部分使用着，倒是不会发生程序崩溃，但是有可能破坏了潜在的重要数据。这样的话可能只是破坏了图像，也可能是AI的奇怪操作，还可能是程序崩溃导致蓝屏，或许是更坏的情况发生。每次做出改变以后，这些奇怪的行为也会随之变化，甚至暂时消失掉。即使添加简单的printf来跟踪错误也可能引起问题自身的改变。毫无疑问，这是程序员遇到的最头疼也最耗时的错误。可以做出的任何可以避免这些错误发生的努力，都可以算作很大的贡献。

性能方面怎么样呢？如果字符数组使用起来这么困难，那么至少必须性能高效。不幸的是，效率不高。要得到字符数组的长度就需要计算遇到空字符前的所有字符数目，因此效率不是特别快。除非人们有先见之明，给数组分配了足够大的空间，否则在连接两个字符串时通常需要分配新的内存来保存两个字符串的备份。另外，为了避免空间溢出，数组总是比需要容纳的字符串大，所以在存储内存方面字符数组也不是太好。

9.3.2 字符串类

C++具有弥补这些不足的手段，在STL里引入了string类。字符串强调字符数组的所有主要问题。

- 字符串会根据需要增长，因此程序在写入字符串时，永远不会超过空间或者产生缓存器溢出。
- 不必担心与字符串长度相应的内存分配问题。

另外，string类提供了许多先前由C函数库提供的字符串操作功能。这样就形成一个优势，即可以把字符串视为对象，进而语法更清晰，同时也更符合使用习惯。

```
string text1 = "This is an example string";
string text2 = text1;                    //We can copy them
string text3 = text1+text2;              // Append them
if(text3 == text2)                       //Compare them
  //...
```

但是 string 如何实现与 C 语言风格的字符数组匹配呢？很高兴的是，设计 string 类的人心中清楚意识到这一点，因此可以轻易地使用 string 类中的 c_str 函数，完成字符串到字符数组的变换。返回的字符指针可以像任何字符数组一样使用。

```
string text = "Yada, yada,yada…";
char oldstyle[256];
strcpy ( oldstyle, text.c_str() );
```

需要注意的一点是 c_str 返回一个 const char 指针。因为这个常量仍然允许程序的其他部分在不了解字符串的情况下来更改字符串的内容，所以该常量不应该被强制转换。如果需要从字符数组转换到字符串，那么就需要创建一个新字符串，或者把数组的内容复制到一个已经存在的字符串里。这样虽然会造成复制字符数组内容的情况，但却是最安全的方法。

这里没有提到的一个操作就是如何写入一个字符串，不仅仅是添加一个字符串，通过拆分和连接字符串就可以很容易实现这个功能。写入一个整数，浮点数或者其他类型的数据就稍微有点复杂。如果使用字符数组，就可以使用方便的 sprintf 函数来自由地写入。

```
char txt[256];                          // Let's hope it's large enough!
SprintF( txt, "Player %d wins with %d points and an accuracy of %.2f.",
player.GetNumber(), player.GetScore(), player.GetAccuracy() );
```

纯 C++论调者可能会畏缩，但是即使 sprintf 可能是类型不安全，并且看起来也不是太美观，但是它确实可以实现这个作用。我们鼓励使用流来实现。

```
ostringstream oss;
oss<< "Player"<<player.GetNumber() << "wins with" <<
      player.GetScore() << "points and an accuracy of" << player.Get
                                                          Accuracy();
string txt = oss.str();
```

这个版本是类型安全的，并且可以实现同样的功能。不幸的是，指定数据的格式不是太直观，因此在前面的例子里，忽略了游戏者要求的小数点后面两位数字的精度要求。

最大的缺点就是为了进行输入和输出，需要引进流。这样做是好的，并且对于流操作是有许多好处的，但是并不是每个人都准备使用它。当需要向字符串中写入的时候，没有流的字符串就显得有点欠缺。

一个可选方法就是像前面做得那样，写到字符数组里面，接着把数组转换成字符串。可以说这不是最好的解决办法，这样在数组到字符串的转换中会引起额外的性能损害，不过这个方法确实可以奏效。

所幸的是，Boost 库提供了新的选择，即格式库 Format。格式库提供了利用格式字符串格式化字符串内容的能力。这里有几个变种，但是格式化字符串支持的类型之一就是 printf 风格的，许多程序员对它已经是非常熟悉了。最后可以像以前操作字符数组那样，直接向字符串写入，惟一的不同就是现在使用的分隔符是百分号而不是逗号。

```
string txt = boost::io::str(
             format("Player %d wins with %d points and an accuracy of %.2f."%
                    player.GetNumber() %
                    player.GetScore() %
                    player.GetAccuracy() ));
```

直到现在，可能还想知道为什么 string 类是 STL 的一部分。从目前所见到的来看，string 类不像一个模板。实际上它就是一个模板。string 类是一个 typedef 定义的模板：

```
typedef basic_string<char> string;
```

模板名称是 basic_string，它可以作用于字符或者其他类型的元素。例如，另外一个有用的字符串类型就是包含 wide 字符（wchar_t）的字符类型，当游戏在世界的不同地方使用，碰到与我们的字符集不同的字符时，这个类型在本地化游戏中就显得非常有用。对这种类型的字符也有一个 typedef 定义，称为适用于 wide-character 字符串的 wstring。

从技术角度讲，与 vector 和链表一样，字符串也只是一个容器。然而，既然字符串有完全不同的使用目的，所以在本书中特意列为了单独的模板类型。

9.3.3 性能

如果以前使用字符指针来传递字符串，那么就需要适应使用 string。为了将文本作为函数参数传递，在使用字符指针实现时，仅仅传递了字符指针本身。这样的操作因为只复制指针本身，所以操作性能很好。用字符串实现的话，就略微有些不同；我们必须留意的是如何把字符串作为参数传递。如果只是传递字符串值，则会创建一个字符串对象及其内容的备份，这会造成内存和性能方面的影响。为避免这种情况，在不需要函数更改字符串内容时，多数情况下应该通过引用，同时可能是 const 引用来传递字符串。

同样，这样的操作也适用于函数的返回字符串，如果把函数的返回值指定为 const 引用，那么就避免了形成额外的复制操作。否则，如果把返回值指定为一个新的字符串，那么字符串的内容就要执行复制操作。以下的代码表示了以上两种情况。

```
//Bad code! We are making two copies of the contents of the string!
void SetHighScore(string name);

string playerName = player.GetName();
```

```
SetHighScore(playerName);

//Much better version. No copies involved.
Void SetHighScore (const string & name);
Const string & playerName = player.GetName();
SetHighScore(playerName);
```

另外一个常见的性能损耗在上一节已经提到过：每当把字符数组转化为字符串时，就包括分配新的缓冲区，然后把所有的字符复制到字符串中。如果原始数组不改变的话，这样的方式可能就显得不必要了，但是大多数实现仍然是采用这样的方式来复制内容。在同时使用字符数组和字符串的代码中，这样的方式就会成为一个大问题。所幸的是，把字符串转换为字符数组是非常方便的。

即使只使用字符串，把一串字母作为函数参数通常也会引起字符串内容向字符串对象的复制。例如：

```
void SetPlayerName(Const string & name);
//Will usually cause a new string buffer to be created and the
//contents of the literal to be copied to it, passed to the
//function, and then discarded. Pretty wasteful!
SetPlayerName("Player1");
```

另外一方面，在游戏中不应有过多硬编码的字符串。大多数这类数据应该从美工和设计者生成的源字符串中读入。尤其是像文本这样的内容，更应该采用从外部资源里读取，这样更有利于实现程序的本地化。

实际上，在自己的 STL 实现中，通过值的形式来传递参数，可能不会有太大的性能损耗。需要记住，不论好坏，即使完全遵循规范进行，在不同的 STL 实现之间也存在着很大的区别。因此自己的实现要尽可能使用引用计数和 copy-on-write(CoW)。不幸的是，即使引用计数可以更有效地复制字符串，其自身也存在可能引起完全不可预知结果的倾向，这种情况在多线程环境中更明显。如果自己的字符串实现使用了引用计数，而想使这一特点不起作用，可以使用别的 STL 实现或者考虑使用本节末尾提供的替换方法。

9.3.4 内存

如果在开发平台内存比较小的情况下使用了许多字符串，那么内存使用就可能成为主要的关心问题。为了使得字符串连接操作在不对字符串进行重新分配和复制的基础上有效执行，每个字符串都分配有额外的填充位。不幸的是，如果不计划增加字符串的大小，这样的做法只会导致空间的浪费。具体分配了多少空间用于填充位实际上是与实现相关的，但是通常是 16 位或者 32 位，在通常情况下这是不需要担心的，但是如果使用了成千上万

个字符串，就需要考虑它们造成的累加效果了。

更有趣的是，一些STL实现也使用了内存共池模式。这意味着分配几个字符串不会引起内存分配，但是同时也意味着释放一些字符串也不会释放这些字符串所占用的所有内存。多数时候这样的做法能允许更快速的字符串操作，所以是有利的。但是从另外一个方面来看，如果内存已经显得紧张了，那么这么做就会使情况变糟，尤其是在程序执行过程中的某些环节（可能是在关卡装载的过程中），你的字符串使用会明显提高，然后再回到正常水平。在这个时候，预留给字符串的内容数量就会比实际需要的大得多。

另外，每个字符串有一个固定的系统开销。这也取决于实现，系统开销可能从4到16字节不等。这些额外的字节用于保留基本信息，例如字符串的开始地址或者长度。

9.3.5 可选方法

如果string类不能满足用户需要，那么在选择使用char *之前，还有几个选择可供使用。Rope、vector<char>和CString类都是值得考虑的选择。

1. Rope

Rope类是第一个选择。这个类不是标准的组成部分，但是却通过几个实现提供了出来。正如string类一样，它也是一个模板类，因此就有可能来指定要处理的元素类型以及指定分配算符：rope<type,alloc>。

Rope类的目的是把大的字符串作为一个单元进行对待。实现的方法是把字符串存储在几个内存块里，而不是一个连续的内存里。尤其对于大字符串的分配、连接以及子串操作是非常快的。另外一方面，在Rope上处理单个的字符开销就很大。同时与string类相比，在rope中实现字符串向字符数组的转化也是开销很大。很显然，Rope类不是完全作为string类的替代品，但是在有些特定的场合下，却是值得考虑使用的。

2. vector<char>

字符串与一个字符序列容器没有差别，我们对vector操作的语法、性能以及内存使用特点已经非常熟悉，那么为什么不使用vector<char>呢？因此这也是一个不错的候选。

这里没有显式的调用来实现向字符数组的转换，但是使用一点小技巧就可以得到指向存储字符的序列内存块指针。只要进行的所有操作都保证在字符串末尾保留NULL空字符，那么一切都会正常运行。

vector<char>的主要优点是没有引用计数或者CoW，因此可以确保文本数据的安全。对于不想自己编写string实现，而又不想转向STL的时候，这会是最好的选择。

该方法的主要不足是缺少许多字符串相关函数，例如替换子串操作。根据一些基本STL算法和vector函数，有可能对这些功能全部实现，但是却需要自己动手编写，另外，最终给出的操作语法可能与人们习惯的string语法不同。

3. CString

微软基础类库（MFC）为了解决 char *数组所有的操作问题，引入了 CString 类。这个类与 STL 中 string 类相比，有着不同的发展方向，因而与 STL 不兼容。但是它却在 string 类问世以前许多年就形成了。

因为这个类在创建的时候没有对模板的正规支持，所以 CString 不是一个基于模板的类，但是这个类却可以完成 string 类的一切功能。如果说有什么区别的话，确实还有一些不同，那就是该类经常受到责备的地方，即过于现实化的方法和为数太多的接口。

对于 CString 类最大的抱怨就是与 MFC 和 Windows 平台的依赖。这么说也不完全正确，因为有方法可以允许用户在不引入 MFC 头文件或者库函数的情况下使用这个类，但是仍然需要依赖于 Win32 API。这样的话，如果用户的目标平台不是基于 Windows 的平台，则仅仅这一点就把这个类排除在外了。

即使是使用 Windows 平台进行开发，除非依赖大量的 CString 编码，否则没有理由不选择标准、轻便的 STL string 类。

9.4 分配算符（高级话题）

有可能出现这样的情况，即 STL 的内存分配策略不能很好的满足需要。原因可能是多种多样的。可能因为在处理基于结点的容器时，频繁的分配和释放内存造成内存碎片或者引起严重的性能拥塞；也可能是像在第 6 章看到的那样，需要更好的对内存的使用进行管理；还可能是为了提高数据缓存的一致性，需要将容器中的元素在内存中保持连续；还可能是意图在堆栈中临时创建元素，并在没有任何损失的情况下完成释放。一旦遇到这些情况，解决方法就是使用定制分配算符。

留意上一章中使用的限定词："可能碰到这样的情况"与"如果这样的情况发生……"，是否需要改变内存分配策略，是依赖于用户使用的平台和自身的需求的，而不要机械地认为 STL 内存分配存在问题。假如这样的话，就要依靠自己的运气了。有些时候，细细阅读本节应当是一个不错的主意，因为这对认识在需要编写自己的分配符时，遇到的潜在问题及其解决方法是有益的。

定制分配符的核心就是在需要的时候，为一个对象分配和释放内存。尽管听起来与在类中重载 new 和 delete 很像，但是确实不是一回事。遗憾的是，伴随着分配符还真的带来新问题，并且可能由于编译器缺乏支持，所以情况会变得更糟。

实际上，分配符针对不同的平台，应该在实现手段上也稍有差别。因此，在 CD-ROM 里没有给出一个全面的、可工作的分配符。可替代的最好方法是从自己的 STL 实现中的默认分配符开始，来编写定制分配符，修改与新内存分配策略相关的特定部分。

现在给出一个例子，像在第 6 章讲述的例子那样，为了更好地跟踪内存的分配情况，开始用自己建立的内存堆栈来创建分配符。每当进行一个分配操作，就要调用与堆栈对应

的操作符 new 的新版本。

那么要找出标准的 STL 分配符，然后把它们复制到别的地方，并尝试更改分配和存储单元分配函数来调用重载的 new 和 delete 运算符。这可能是最初的尝试：

```
template<typename T>
MyHeapAllocator{
public:
   pointer allocate(size_type nNumObjects, const void * hint = 0){
         return static_cast<pointer>(new(nNumObjects*sizeof(T),pHeap));
   }
   void deallocate(pointer p, size_type nNumobjects){
         delete(p, nNumObjects* sizeof(T));
   }
   //…
};
```

这里有一个小问题。如何把 pHeap 传递给运算符？理想点讲，愿意每次遇到一个分配的时候，让分配符保持一个指向堆栈的指针，而这个堆栈是我们想使用并且想把它传递给 new 函数的。这里引出了 STL 分配符的第一个怪事：分配符不应该拥有任何使用对象的状态。因此不能保持指向堆栈的指针。为了克服这个问题，需要围绕堆栈创建一个新的封装类，以便将这个类作为参数传递到分配符模板中。以下是编写分配符的新尝试：

```
class HeapGraphics{
public:
   static void * allocate(size_t nNumbytes){
         return new(nNumBytes, a_pGraphicsHeap);
   }
   static void deallocate(void * p, size_t nNumBytes){
         delete(p,nNumBytes);
   }
};

template<typename T, typename Heap>
MyHeapAllocator{
public:
   pointer allocate(size_type nNumObjects, const void* hint = 0){
         return static_cast<pointer>(Heap::allocate(
               nNumObjects*sizeof(T),pHeap));
   }
   void deallocate(pointer p, size_type nNumObjects){
```

```
        Heap::deallocate ( p, nNumObjects*sizeof(T));
    }
    //...
};
```

这个新版本考虑所有的分配符规则，并允许从自己的堆栈分配所有内存。惟一的不足是需要为想使用的每个堆栈创建一个新的封装类，这样是显得有些麻烦。然而正如已经提到过的，STL 分配符不是完美的。

从用户的角度看，分配符内部的实际工作是有点棘手，但是这些确实是在编写自己的定制分配符时需要掌握的。可参照本章最后列出的阅读建议，相信会得到自己感兴趣的书目。

9.5 当 STL 不满足要求时（高级话题）

在占用了几章的篇幅讨论 STL 如何有利于游戏项目之后，具有讽刺意味的是，要讨论可选内容，而将 STL 搁置一边。必须非常清楚一件事情：STL 是一个重要的开始点。几乎没有不使用 STL 的理由。如果小心使用，正如在本章和第 8 章曾提到的，你将开始在坚实的数据结构和灵活的算法基础上开发游戏或开发工具。在微不足道的工作上花费时间越少，就可把越多的时间用在提高游戏性，研究新的技术，甚至是做出技术突破方面。但是如果你在开发 PC 游戏，那么这些可能是必须要考虑的事情。

然而，尤其是与 PC 相比，在只有有限资源下进行游戏机终端上的游戏开发的时候，可能遇到 STL 不太适合的情况，常见的问题就是对内存分配缺乏很好的控制。大家可能已经摆弄过创建定制分配符这样的工作，而且还仍然不太满意 STL 容器的内存分配模式。也有可能大家想可靠地释放 vector 中的内存，或者想更多地对 deque 的块尺寸实行控制。乐观地说，我们是看到了妨碍游戏的东西，并且是这些东西推动着这样做的。不要认为更多地控制内存就是好事；如果没有以任何方式对游戏形成不利影响，那么不要在这些方面浪费时间，只有在游戏中至关紧要的时刻，发现游戏慢了下来，或者自己的游戏比想象中使用了更多内存的时候，才需要寻找可替换的实现方法。

关于 STL 的另外一个常见抱怨，一般也是对模板的抱怨，就是代码膨胀。与其他的事情相比，有些编译器更容易导致代码膨胀，创建同一个在不同类中反复使用的模板，有可能引起最终执行文件大小的快速增长。

最后，到目前为止没有考虑的另外一个原因就是编译时间的增加。不要忘记 STL 是由模板组成的，而模板只是编译时刻产生的代码，因而就会增加编译器的工作。编译器做的工作越多，编译步骤就要花费越多时间。对游戏的细小改动都需要很长的编译时间，花费越大而且开发也越麻烦。如果出现这种情形，并且怀疑是由 STL 使用过度引起的，那么就应该考虑使用其他方法。然而，首先要确保确实是 STL 的原因。许多项目开发者不关心他们的物理结构和头文件的包含方式（参见第 15 章处理大型项目）。

碰到这样的情形应该怎么做呢？有更好的库可以使用吗？遗憾的是，到目前为止还没有这样的库。大概最好的方法就是提供自己定制的容器了。

在听到了这么多关于 STL 的优点之后，再听到推荐写自己的容器这样的话，可能有点吃惊，但是遇到了这样的情况，那么这可能是最好的解决方法了。但是不要完全放弃 STL 来开始写自己的容器类。

看一下自己的代码的主要部分，考虑一下自己最常使用的 STL 部分，或者哪些部分自己感到费神。90%的 STL 使用机会都局限在一些容器上，可能是 vector、一些 map 或许是字符串。这不意味着不使用 STL 的其他部分，而是使用以上所述的容器可能是自己日常 STL 使用中最常见的。

一个好的方法是只替换常用的容器，或者是自己不太清楚的容器。自己仍然可以在编码中正常使用 STL 中的其他内容，而不需要完全撇开 STL。例如，可以创建自己版本的 vector，并命名为 myvector，或者是 array，或者是自己想叫的任何名字。这个 vector 可以拥有一个与 STL vector 相似的接口：仍然有 push_back、clear、resize 和 reserve 函数。但是也可以增加新的函数来更好地对内存分配和释放进行控制，还可以提供自己容器的定制版本来满足自己的需要。例如，可以提供一个 vector，该 vector 仅包含最基本的 vector 内容，以求内存消耗最少，以便自己游戏中的所有游戏实体，也许成千上万个，全部使用它。

还可以提供 iterator 来访问容器，如果关心这些 iterator 如何实现的话，仍旧可以在自己的容器中，使用 STL 算法的相当部分。因此在某种程度上，不是完全替代了 STL，而是对它进行扩充，并且在自己新的容器里也可以使用 STL 某些部分。

另外，这不是一个容易理解的工作。不能低估创建一个适合自己需要，并且比 STL 版本更有效工作的健壮的、可靠的容器需要付出多么大的努力。当决定是否真正需要以及究竟有什么不同需要的时候请花点时间考虑一下，否则就继续使用 STL。如果这样的话，那么可能永远都不会自己创建新容器。

9.6　结　论

本章介绍了 STL 的另外一部分内容。算符作为 STL 中的重要组成部分，可以以比使用函数指针更灵活的方式传递函数，从算符开始介绍，接着讲解了 STL 提供的 4 种类型的算法。

- 无改变算法，一类作用在元素序列，但不改变其内容的算法。用于查询元素，对元素计数，元素遍历等的执行。
- 有改变算法，是一类修改序列中的元素，但是不对元素进行排序的算法。用于一定范围内元素的复制，删除或者改变元素的顺序。
- 排序算法，一类对一系列元素进行排序并执行其他相关操作的算法。
- 数值算法，只是少量可以对一定范围内的元素执行数值计算的算法。

其次介绍了 STL 中的字符串类及其对应部分，即用于 wide 字符的 wstring 类。尽管它

们看起来有点怪异，但是在处理文本字符串的时候却可以更简单的实现，与使用 char*相比，多数时候它们是较好的方法。

最后，简要讲述了在需要 STL 默认提供以外功能的时候，如何对 STL 进行处理。首先介绍了如何为自己的容器编写定制分配符，以及希望从中得到什么，然后分析了在什么情形下需要编写自己的容器代码而不是使用 STL 的容器。

9.7 阅读建议

以下的书目中包括许多综合性的主题。与本章相关，这些参考书中包含许多关于算符方面的好建议，对字符串、vector<char>以及 STL 算法的论述：

Meyers, Scott, Effective STL, Addison-Wesley,2001.

接下来是一些非常优秀的通用算法方面相关的书籍。可以发现许多 STL 中的算法在这些书本里进行了详尽的解释，包括它们是如何实现的，以及它们有什么样的结果。顺便提一下，也会找到在 STL 容器中用到的大多数数据结构的较好的描述。

Cormen, Thomas H., et al.,Introduction to Algorithms, McGraw-Hill, 1990.

Knuth, Donald E., The Art of Computer Programming, 3rd ed., Addison-Wesley,1997.

下面是有关字符串类及其实现以及相关库函数方面比较好的资源。

Strings in SGI STL, http://www.sgi.com/tech/stl/string_discussion.html.

C++ Boost Format library, http://www.boost.org/libs/format/index.htm.

分配算符是一个技巧性很强的主题。可能是因为分配算符是过于依赖平台，所以关于这方面没有太多的参考书。以下是一些好的入门书目：

Meyers,Scott,Effective STL, Addison-Wesley.2001.

Treglia, Dante, et al., Game Programming Gems 3. Charles River Media, 2002.

Josuttis, Nicolai M., The C++ Standard Library. Addison-Wesley,1999.

Josuttis, Nicolai M., User-Defined Allocator, http://www.josuttis.com/ cppcode/ allocator.html.

另外，以下参考部分给出了关于 STL 方面非常全面的介绍：

SGI's Standard Template Library Programmer's Guide, http://www.sgi.com/tech/stl/.

第3部分　专门技术



除了编写C++代码并使其快速执行以外，还有很多使游戏成为一个整体的技术。代码装配的方式对于项目的其他部分有重要影响。游戏开发中，有一些特定任务会在每个游戏中重复出现：创建对象，保存游戏状态，为工具编写插件等。寻找最有效的方式来实现所有这些任务，可能需要花费许多年的反复试验。

本书的后半部分试图对实际游戏开发中证明行之有效的特定技术做出介绍。通常，每章不只讲述一种技术，并同时会讨论不同技术的优点。通过学习这些技术，读者可以将这些技术转化为自身的技能，从而在自己的项目中运用它们。

第10章 抽象接口

本章内容：

- 抽象接口
- 通用C++实现
- 作为绝缘层的抽象接口
- 作为类特征的抽象接口
- 其他方面

抽象接口是一个非常有用的概念。它允许把类实现和类接口彻底分离。这就为我们骤然打开了一道大门，这道大门使以前许多不可能的事情成为了可能。例如在运行期改变程序的表现，在程序发售以后扩展程序功能，或者通过实现特定特征来赋予类不同的表现。

本章解释在C++里如何实现抽象接口，并展示在游戏和工具中如何发挥它们的优势。第11章将给出抽象接口插件方面的一个特例。

10.1 抽象接口

类的主要目的就是表述一个概念，对用户而言，它封装了类的具体实现方法以及用户无关的内部实现细节。理想的讲，在类的外部，只有描述类自身功能的函数接口是可见的。

遗憾的是，C++并不十分彻底。毫无疑问，一部分原因要归咎于它是对C的继承，C++向外界提供的就不只是公共接口了。一个C++类通过一个头文件来描述，任何需要使用这些类的程序都需要包括这个头文件。但是在一个头文件里，往往不只包含类接口信息，还包括其他正确编译必需的头文件，自身所有的保护的、私有的函数声明，以及所有成员变量列表。当然，编译器只允许程序访问类的公有函数和成员变量，但是关于实现细节方面的信息会暴露给程序的其他部分。

为什么抽象接口会显得很重要呢？一般情况下，信息暴露不是一个大问题。毕竟，抽象接口关系到从开始如何来设计类，并且也确实相当有用。然而，有时却需要更好的封装。通过减少类的实现和使用类的代码之间的耦合性，就可以得到以下的好处。

- 只要一个重编译和重链接步骤，就可以在无须调整任何其他代码的条件下更改类的实现。设想能够从几个不同的实现中选择其中一个来测试和权衡，并决定最适合当前游戏的最好实现，或者是尝试几个空间分割算法（octrees、quadtrees、BSPtrees等）来替换类的实现。

- 可以在运行时刻改变类实现，这把事情更深入了一步，但是仍然可能很好地分离类的接口和实现。对于根据用户硬件，或者根据菜单选项来决定使用什么渲染系统这样的情况，就显得极其有用。
- 能够在游戏发布以后提供新的实现。设想发售了游戏，然后又通过增加新的单元、新的行为或者新的游戏类型对游戏进行了扩展，那么这些扩展部分都可以作为新的下载处理对待。如果程序从开始就是这么设计的，那么就有可能实现这样的功能。同样地，这些方法也可以应用在工具中（无论是内部开发游戏使用的工具还是与游戏一起发布的工具），可以在不需强迫发布新的可执行程序前提下，通过插件对工具实现扩展（要详细了解可以参考第 11 章的插件）。

抽象接口就是 C++中的一类特殊组织方式，通过这种方式可以实现接口和程序实现部分之间的分离。只要稍加留意事物的组织方式，就可以利用抽象接口实现前面提到的目标。

除这些特殊目的之外，如何降低游戏代码和它控制的对象的耦合性也是一个值得努力的目标。与直接引用对象相比，抽象接口表达更清晰，代码也更易维护。同时也在不同代码之间划定了更明确的分界线，这就允许一个团队的几个程序员更容易地并行处理相同的代码。如果缺乏这样的特性，程序员通常只能串行工作且必须经常交流彼此的工作进度。

抽象接口不是一个新概念，也不是一个只适用于游戏开发的概念，它是一项有用的通用编程技术。一些公司已经基于抽象接口的概念生成了自己的 API 函数，甚至还提供了他们自己的标准抽象接口函数和宏。微软的 COM 就是这样一类主要依赖于抽象接口的 API。事实上，COM 可以完成本章中所要开发的一切事情。

那么，为什么要花时间来编写自己的抽象对象接口而不是直接使用 COM 呢？主要的原因就是，在大多数时候只是想使用最基本的功能，可能不计划使用远程过程调用或者 COM（以及它的后继，例如 COM+）提供的其他任何高级特性，所以也就不需要任何其他的包袱。另外，抽象接口作为一个非常简单的概念，也是值得单独学习的。如果在将来意识到需要使用 COM，那么也可以通过学习抽象接口技术来更好地理解如何有效地使用 COM。最后，对使用 COM 而言，最主要的一个不利因素就是它的平台相关性。目前，不存在可以主宰游戏市场的单一平台。PC 机游戏只是游戏市场的一小部分，而大部分市场被游戏终端（也就是各种游戏机）占领着。即使对游戏终端而言，也有几个完全不同的平台。如果生成自己的平台无关的抽象接口版本，那么就可以很容易地在任何具备 C++编译器的平台上重用这些代码。

10.2　通用 C++实现

C++中的抽象接口就是一个只有纯虚函数的类，没有具体的实现，没有成员变量，没有其他任何东西。标记这些纯虚函数的方法是在它们的声明末尾添加一个=0，这就表明，如果想要创建这个类的对象，那么派生类必须提供类实现。一个简单的抽象接口类如下所示：

```
//IAbstractInterfaceA.h
class IAbstractInterfaceA{
public:
   virtual ~ IAbstractInterfaceA() {};
   virtual void SomeFunction() = 0;
   virtual bool IsDone() = 0;
};
```

因为抽象接口类不需要有任何类型的实现，所以甚至不需要相应的.cpp 文件。抽象接口类只是对接口的一个简单描述。注意类名称的前缀是 I，这表明该类是一个抽象接口。当然不是必须使用这个前缀，但是使用了它确实是一个有益的提示，提示当前使用的类是一个抽象接口类。为了创建基于这些接口的实现，就要继承抽象类，并提供其中所有函数的实现。

```
//MyImplementation.h
class MyImplementation:public IAbstractInterfaceA{
public:
   virtual void SomeFunction();
   virtual bool IsDone();
};
```

注意，在这里，因为将要在.cpp 文件中提供函数的实现，所以，这里的函数不再是纯虚函数。

```
// MyImplementation.cpp
void MyImplementation::SomeFunction(){
   //…
}

bool MyImplementation::IsDone(){
   //…
   return true;
}
```

到这里为止，任何人都不需要考虑该类究竟使用了什么特定的实现，就可以在程序里选择并使用它了。

```
IAbstractInterfaceA * pInterface = new MyImplementation;
//…
```

```
pInterface->SomeFunction();
if(pInterface->IsDone());          //etc…
```

这只是一个一般情形，所以看起来还不是很有用。感觉好像是毫无理由去额外增加了一个间接层，使得程序不够清晰并且也难于维护。其实，如果可以更好地利用这个中间层，那么就可以看到其作用了。现在仔细看一些有关抽象接口的例子，以及在游戏中如何有效地使用它们。与许多事情一样，细节往往是最难的部分，因此将描述具体的实现并体会如何灵活地解决问题。

10.3　作为绝缘层的抽象接口

首先，抽象接口最直观的用途就是简单地充当一个类实现与程序其他部分之间的绝缘层。看一个例子。

创建一个布局（layout，相当于游戏 GUI），这样的布局也需要渲染。这里会有两类图形渲染器（渲染 API）：OpenGL 和 Direct3D。显然，根据自己使用的平台和程序需要，也可以进行扩展，以便包括其他渲染类型。我们的目标是保证使用渲染的程序不必知道当前使用的是什么渲染方式，并在不做任何调整的基础上，使用任一类渲染工作。另外，还希望在运行期能够切换渲染器。

这是一个有关抽象接口方面非常好的应用例子。接下来的是用于图形渲染器的 C++源代码：

```
//GraphicsRenderer.h
//This is the abstract interface class
class IGraphicsRenderer{
public:
   virtual ~IGraphicsRenderer() {};
   virtual void SetWorldMatrix(const Matrix4d & mat) = 0;
   virtual void RenderMesh(const Mesh & mes) = 0;
   //…
};
```

这里仅仅给出了两个典型函数。在实际情况中，可能需要渲染器在处理材质，渲染状态、阴影、光线等方面的其他函数。现在给出基于以上接口的两个实现：一个用于 D3D，一个用于 OpenGL。

```
//GraphicsRendererOGL.h
#include "GraphicsRenderer.h"
#include<gl.h>
```

```
class GraphicsRendererOGL:public IGraphicsRenderer{
public:
   virtual void SetWorldMatrix(const Matrix4d & mat);
   virtual void RenderMesh(const Mesh & mesh);
   //…
}

//GraphicsRendererD3D.h
#include "GraphicsRenderer.h"
#include <d3d.h>
class GraphicsRendererD3D:public IGraphicsRenderer{
public:
  virtual void SetWorldMatrix(const Matrix4d & mat);
  virtual void RenderMesh(const Mesh & mesh);
  //…
}
```

在它们相应的.cpp 文件里面，提供了抽象接口中各函数的实现。只要程序通过 IGraphicsRenderer 指针使用图形渲染器，就可以很容易地改变具体实现。我们需要做的只是创建需要类型的渲染器，然后把指向渲染器的指针传递到程序中使用就可以了。

```
IGraphicsRenderer* g_pRenderer=new GraphicsRendererOGL();
//…

//Now we don't care which renderer we're using
g_pRenderer->SetWorldMatrix(ObjectToWorld);
g_pRenderer->RenderMesh(mesh);
```

在运行期，只要创建一个新的对象，并通知系统使用这个新对象而不是原来的旧对象，就可以自由改变使用的渲染器类型了。这里惟一的难点就是在渲染器内部对诸如视频模式、纹理等已编好的设置做必要调整时有点困难。要是通过抽象接口角度对待的话，这些操作就很容易实现。

10.3.1 头文件和厂

用前面讲到的方式来组织渲染器还会有一个额外的好处。可以看出，GraphicsRendererOGL.h 和 GraphicsRendererD3D.h 两个头文件包含了一个（也可能是多个）平台相关的头文件。有关特定实现方面的头文件包含了实现的具体细节，可能包括专门针对 OpenGL 或者 Direct3D 使用的结构、枚举以及宏，因此必须包含这个头文件。

如果图形渲染器的设计中没有采用抽象接口，那么程序就需要直接使用 OpenGL 或者 Direct3D 渲染器了，这就意味着程序的其余部分也要被迫包含与渲染器相关的这个平台相关头文件。虽然这也不是什么太大的坏事，但是却有以下几个缺点。

- 这些头文件通常比较大，它们可能还包含了其他的头文件，这样就会有相关的结构、类或者宏。如果每个使用渲染器的文件都必须包含这个头文件，编译时间会明显增加。而使用抽象接口，既为编程提供了清晰的设计，又保证了更快的编译速度。这是典型的双赢情形。如果利用了带有预编译头文件支持或者一次包括许多头文件的其他方法的话，编译时间还会进一步减少。不幸的是，这样的方法却往往使下面提到的这个不足更为明显。
- 包含平台相关渲染器文件的任何代码都可以访问渲染器的任意内容。这样就容易导致程序中与渲染器无关的部分都会依赖 OpenGL 或者 Direct3D，并且只有在转换到其他不同平台的时候才可能发现这个问题。从这一点讲，就会花费太多时间，造成许多麻烦，甚至可能是延误最终期限。有可能使用到碰巧在 d3d.h 文件中定义的一个宏，接着使用了一个 typedef，紧接着调用一个辅助函数来建立矩阵等。在了解这些以前，因为自己没有明确的在程序中包含 Direct3D 头文件，所以代码是在我们毫无意识的情况下被全部锁定到一个特定的平台。

在第 15 章（处理大型项目）中，会对项目的物理结构进行讨论，在那里可以看到有关这些方面的更详细的讨论。然而，为了能够创建对象并向程序其余部分传递对象，一部分代码需要包含这些实现头文件。这些可以通过限制到单独一个.cpp 文件的方式来实现，因此通常不是问题。

更深入一步，利用厂把实现进行封装。一个厂是允许创建相关对象的简单设计模式。在讲述的例子中，在不需要实现的头文件情形下，厂可以生成渲染器的不同实现。

以下的代码展示了厂创建渲染器的方式。可以注意到，我们不是直接处理 GraphicsRendererOGL 和 GraphicsRendererD3D 的特定类实现，而是通过名字请求了厂函数。

```
GraphicsRendererFactory factory;
IGraphicsRenderer* g_pRenderer;

//This creates an OpenGL renderer
g_pRenderer = factory.CreateRenderer("OpenGL");

//This creates a Direct3D renderer
g_pRenderer = factory.CreateRenderer("Direct3D");
```

实际上，厂的实现是很直观的，它检查传递到 CreateRenderer 函数的名称，并在正确的渲染器实现中创建一个新对象。目前，只有厂类包括每个渲染器需要的实现头文件，而程序的其余部分并不知道这些文件是否存在。

至此，已经彻底完成了接口和实现的分离。正如大家可以想象到的，这种分割对于把

代码分离到另外的库中（静态或者动态库）而言，也是一个非常好的边界。库中需要开发的惟一事情就是抽象接口以及创建新的实现需要的厂，其他的任何内容都可以保持较好的隐藏。

10.3.2 真实环境中的具体细节

这里还有几个有趣的C++实现细节值得谈一下。这些细节不是关于抽象接口概念方面的，而是涉及如何以更有效地方式在C++中实现抽象接口方面的。

第一点需要提到的，就是在抽象接口中的所有函数都是虚函数。显然，想通过抽象接口的指针来调用派生类的函数，抽象接口成了全部的切入点，因此必须把抽象接口的所有函数声明为虚函数。

除了声明为虚函数以外，还必须保证是纯虚函数（利用紧接着函数声明的=0标志是纯虚的）。这样做以后，除非有人提供了这些函数的实现，否则编译器不允许创建这种类型的对象。不是必须把这些函数声明为纯虚的，但是可能这是一个好的暗示，提示需要对接口要求的全部函数提供实现。

在上面讲到的例子中，尽管没有对虚析构函数做任何评论，但是读者可能已经注意到了它的形式。现在，大家注意到这个析构函数中略微不同的地方了吗？如果在前面没有留意，那么现在再次给出抽象接口中的析构函数：

```
class IGraphicsRenderer{
public:
   virtual ~IGraphicsRenderer() {};
   //…
};
```

首先注意到的第一点，就是抽象函数自身具有析构声明。并且不止是具有声明，而且还是一个虚析构声明。之所以做出这样的声明，理由与其他虚函数声明是一样的。有些时候，程序很可能把通过抽象接口指针指向的对象删除，如果析构函数不是虚函数的话，程序便调用抽象接口的虚析构函数，这样就不会调用导出类的任何析构函数。而通过把析构函数声明为虚函数，就可以确保调用了正在使用的特定实现相关的析构函数。

另外有趣的一点就是，与其他函数不同，析构函数不是纯虚的。事实上，析构函数右侧的两个花括号就是内嵌的空函数体。利用这样的方式提供了析构的空实现。这样做的话，对编译器显得比较恰当，如果不这样实现，因为析构函数与正常函数工作方式略有不同，所以编译器会出现连接错误。

也可以把析构函数声明为如下形式的纯虚函数：

```
virtual ~IGraphicsRenderer() = 0 {};
```

或者是声明为纯虚函数以后，还可以把空的函数实现体放到.cpp 文件里。所有这些做法功能上是等同的，但是却带有一个缺陷，就是要求所有的派生类必须提供一个析构函数。有些时候可能想这么做，但是也可能不想这么做，因此比较好的解决方法就是不把析构函数声明为纯虚形式，而由派生类决定是否需要自身的析构函数。

最后，使用这种类型的抽象接口经常碰到的一个问题就是，抽象接口可以提供局部实现吗？首先，可能觉得这是一个很奇怪的想法，为什么想到要提供局部实现呢？毕竟，抽象接口的目的就是要把接口和实现分离。遗憾的是，在实际情况中，事物往往不是可以非常清晰地区分出来。有时想要对一个抽象接口形成许多不同的实现，并且在这些实现里只有少数函数是完全相同的。如果抽象接口能够正确实现这样的功能，那么将会显得非常便利。

简单的回答是在有些情况下可以这么做，如果只是一个简单的函数，那么这么做可能是非常好的。同时把这个函数标记为纯虚函数仍然是一个不错的主意，这样就可以迫使导出类显式地调用父类的实现，从而避免了多重继承的不明确性。

当事情变得比较复杂时，如果希望在接口里拥有许多不同的函数实现，那么就需要退回来重新考虑自己的设计了。添加到抽象接口的实现函数越多，抽象接口就越脱离抽象。这样的话，许多希望解决的问题又重新浮现出来，例如，接口和实现的紧密耦合，头文件泄漏到程序的其他部分等。较好的解决方法就是保持纯抽象接口，而从抽象接口派生一个类来提供所有的通用功能，然后从这个中间类生成其他的特定实现（如图 10.1 所示）。

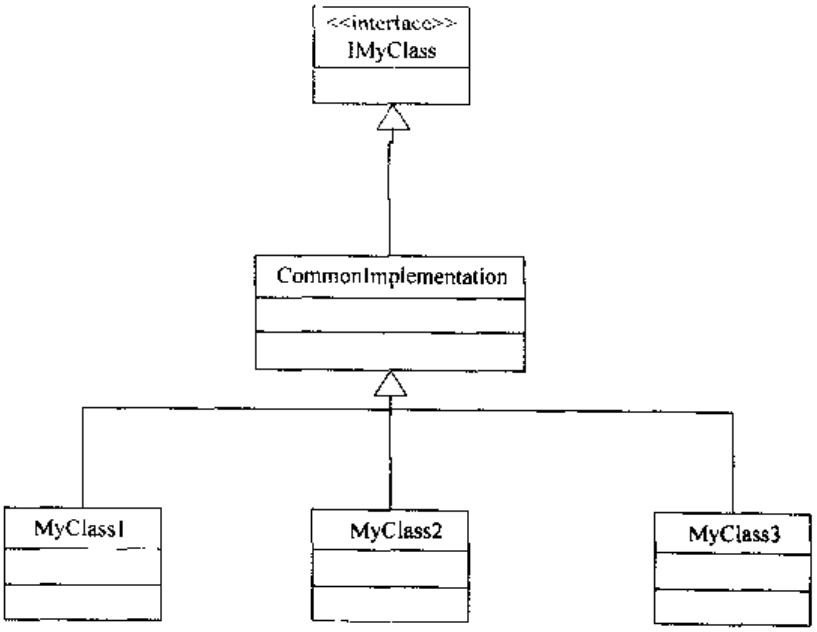


图 10.1　为多个类提供一个通用的抽象接口实现

10.4　作为类特征的抽象接口

10.3 节讨论了如何利用抽象接口来实现接口和实现之间的分离，从而保证比较容易地

变换实现。本节将介绍抽象接口的另外一个方面。在这里，不关心如何来实现转换，而是关注于如何把对象标识隐藏到抽象接口之后，这样就可以使用相同的代码来控制任意类中的对象，甚至是在游戏发售时尚不存在的类。

这里采用的想法很简单：在程序里不固化使用特定类中的任一对象，相反，每次在需要操作执行对象的时候，首先询问对象是否具有与我们的需要相匹配的实现，如果有，就可以调用合适的功能。

例如，考虑一下这样的情形，我们计划对游戏中的所有对象进行渲染。但是，并非所有对象都是可以渲染的，有些对象只是具有单一的逻辑函数，例如转换、计数或者标志等。利用抽象接口，就可以搜索出所有的对象，并得知该对象是否是可渲染的。一旦对象是可渲染对象，就调用渲染函数。这听起来似乎很简单，但是细节方面会显得有点棘手（顺便提及的是，还有其他有关渲染场景方面更有效的方法，而不是检查所有的对象，因此不要认为这个例子就是一个好的架构，只是在这里简单地作为一个例子罢了）。

10.4.1 实现

让我们快速回顾一下不用抽象接口如何实现上面的例子。

1. 固化类型

第一个操作方法就是在游戏中调用代码时固化对象类型，然后遍历所有的对象，检查每个对象的类型。每当找到了一个需要渲染的对象类型，就调用它的渲染函数。然而，从严格意义上讲，这一方法因为固化了特定的对象类型，所以要尽量避免使用。

```
//Pseudo code of a render function using hardwired
// object types
void RenderWorld(){
   for(each object in the world){
      if object is enemy ||
         object is environment object ||
         object is effect ||
         object is terrain ||
         object is sky ||
         //..etc){
                  object.Render();
         }
}
```

这一方法能够工作，但是正如从代码片断里看到的，它显得有些笨拙、容易出错并且不太好维护。每当增加一个新的对象类型，就需要记着改变渲染检查函数，更别提程序中

用相同逻辑来进行其他类型的处理了，例如 AI、碰撞检测以及对象的物理表现等。这种方法基本上不可能在不改变所有代码的基础上新增对象。

2. 由对象决定

比较好的方法是让对象自己决定是否需要渲染。可以就像没有抽象接口一样来做这些事情。所有对象需要做的事情就是实现一个函数，这个函数返回一个布尔值，表明自身是否应该渲染。那么在遍历所有游戏对象的时候，就可以调用这个函数，并根据函数返回结果来决定是否需要渲染这个对象。

```
//Pseudo code of a render function asking each object
void RenderWorld(void){
   for(each object in the world){
         if (object.IsRenderable)
               object.Render();
   }
```

很显然，与前面使用的方法相比，这个方法显得更简洁一些，但是还不是完美的方法。主要的不足就体现在，对每个对象而言，除了实现渲染对象需要的函数（在本例中，就是 Render 函数）之外，还需要实现 IsRenderable 函数。对程序本身来讲，可能不是太大问题，但是当类似这样的特征增加的多时，就需要所有的对象都支持更多的函数。因为这些函数通常为空，并且对特定类而言甚至是毫无意义的，所以就意味着不是每个类都需要这些函数。假定开始为碰撞，操纵或者移动对象添加了类似的函数，那么所有的类就都会带上一些无用的函数了。

更糟的情况，假想已经有许多对象，可能是好几百个不同类型的对象，现在决定增加另外一类特征，例如对象能否发出声音。那么现在就需要调整所有已存在的对象，并创建一个新的函数判定它们是否可以发声。当然，如果所有的游戏对象是从公共的根继承下来的话，实现起来就比较简单，但是事情往往不是这样的（或者是想让其中一半返回 false，而另外一半返回 true）。那么这里就没有更清晰的方法了。

3. 抽象接口

一个更好的解决方法应该是，只声明和实现为了一定类而专门设计的函数。抽象接口和继承允许我们这样做。使用抽象接口实现渲染函数的伪代码如下所示：

```
//Pseudo code of a render function using abstract interfaces
void RenderWorld(void){
   for (each object in the world){
         if( object implements Renderable interface ){
               IRenderable * pRend;
               pRend = object.GetInterface(IRenderable);
```

```
                pRend->Render();
        }
    }
```

不实现特定接口的对象不需要做任何事情。事实上，它甚至不需要知道渲染接口是否存在，这样的方式意味着能够在以后较容易地进行扩展，所以就显得非常有用。

还需注意到的是，一个对象不只可以实现一个抽象接口。接下来创建一个对象，这个对象实现了预期的许多抽象接口，需要做的只是从这些抽象接口导出这个对象，并实现正确的函数。因为对象是从抽象接口继承产生的，所以有关多重继承方面的潜在问题就不再是问题了。这也是在C++中争论是否使用多重继承的一个方面。以下就是为实现 IrenderableI接口而创建的类代码：

```
class IRenderable
{
public:
    bool Render()=0;
    //…
};

class GameEntityPhysical : public GameEntity, public IRenderable
{
public:
    bool Render();
    //…
};
```

10.4.2 查询接口

迄今为止，一直掩盖着一个主要的技术细节：如何判断一个对象是否实现了一个特定的接口呢？一个可能而且有效的方法就是使用 dynamic_cast，正如在第 2 章中介绍的一样，使用强制转换可以确认一个对象是否继承了一个特定的类，如果该对象确实是从指定类中继承过来的，那么就得到了指向父类的指针。这种方式只要在编译选项里选定了 RTTI（运行期类型信息）就可以很好的工作。因为在多数情况下，为了更好地控制内存使用和查询性能，宁愿用自己的系统来替换标准的 RTTI，所以该方式也不总是人们期望使用的方式。

如果那样的话，要替代对 RTTI 的依赖，就要提供一个可以判定对象是否从特定抽象接口派生的函数（这个函数与第 12 章讲述的默认 RTTI 系统会有不同）。在这里要做两件事情。第一件事情就是前面已经讨论过的，判定对象是否实现了特定抽象接口的函数；第二个目标有点微妙，即必须返回一个指向对象的指针，而指针类型是抽象接口类型的。

读者可能回想起在第 2 章里讨论过的多重继承了，使用多重继承的对象把几个虚函数表合并成一个。同时，正确地从一种类型转换到另外类型需要通过偏移量来改变初始指针。调用代码因为不知道被引用对象的类型，所以不能正确地转换指针，这样就应该由对象自己来处理转换过程。执行转换比较方便的场合就是在检查是否提供接口的同一函数里。

在这里，给出函数的一个简单实现。因为是为了回答一个对象是否实现特定接口这样的问题，所以这些函数被称为 QueryInterface。如果实现了接口，那么就返回一个指向接口的正确强制转换指针，否则返回空。

```
void * GameEntityPhysical::QueryInterface(interface interface) const
{
   if(interface == IRENDERABLE){
         IRenderable* pRender=static_cast<IRenderable*>(this);
         return (void*)(pRender);
   }
   return NULL;
}
```

注意在上面的代码中，首先用 this 指针指向想要得到的指针类型，然后作为一个普通 void 类型的指针返回。这样看起来似乎是不必要的步骤，但是转换很可能会改变指针的实际值。缺少这一步的话，返回值就不能安全地转换到正确的接口类型。

很显然，从抽象接口派生出来的每个类都必须实现 QueryInterface 函数。这个函数也需要被不依赖于类定义的程序调用，因此通常情况下，最好的方法是即便这个函数是惟一的成员函数，也要把这个函数放在父类中。

在这个函数里，还假定某处保留了标识每个接口的惟一标识符。在这个例子里，惟一标识符就是 IRENDERABLE。标识符可能使用的是字符串而不是惟一的数值，但是 QueryInterface 可能在每个框架里被调用几百次或者几千次，这样会导致运行时刻的性能较差。为满足程序中的多数需要，简单的数字序列就足够了。如果想让其他人可以对系统进行扩展以及增加新的接口，那么需要提供方法来保证接口标识符的惟一。微软的 COM 系统就是一个非常类似这样的系统，也提供了惟一标识符的方法。

如果创建了许多 QueryInterface 函数，那么就可以简单地把所有功能捆绑到几个宏里，以使其可以更容易地添加到新类中。在使用这些宏以后，创建一个新的 QueryInterface 就很简单了：

```
QUERYINTERFACE_BEGIN
     QUERYINTERFACE_ADD(IRENDERABLE,IRenderable)
QUERYINTERFACE_END
```

添加更多接口只需要在第一个宏之后插入几个 QUERYINTERFACE_ADD 宏。前面例

子中讨论的渲染函数在使用 QueryInterface 以后如下所示：

```
//Render function using QueryInterface
void RenderWorld(){
    foreach object in the world){
        void * pInterface;
        pInterface = object.QueryInterface(IRENDERABLE);
        if(pInterface != NULL){
            IRenderable * pRend;
            pRend = static_cast<IRenderable*><pInterface);
            pRend->Render();
        }
}
```

10.4.3 扩展游戏

在本章开头提到的目标之一就是在游戏发售以后可以进行扩展。那么如何用抽象接口来实现这一目标呢？

首先需要意识到的就是可能根本不需要对代码进行改变。游戏在发售以后进行的许多功能增加，只能利用添加新的数据文件来实现。如果游戏的设计结构主要是数据驱动方式，那么通过提供新的数据文件可以完成许多工作，数据文件包括新的关卡、新的游戏种类、新的游戏角色等。如果计划允许扩展游戏，那么希望是通过提供的工具来修改游戏，而不是编写 C++代码。这样就可以很方便地修改游戏了。

如果有必要发布新数据和新代码，那么有些时候就要替换可执行程序本身或者修补可执行程序。这些更新后的可执行程序可以用最新的类进行编译。因此这种情况下完全没有必要使用抽象接口。然而从开发角度看待的话，是要尽可能地减少游戏代码和它所操纵的游戏对象的耦合性，这样使用抽象接口还是有好处的。

另外一个不同的方法是通过新的游戏部件来发布新的功能，而不是更新可执行程序，当然还可能伴随着使用新数据。在提供了许多不同内容的情况下这样的方式是更可取的，用户只需要下载自己感兴趣的部分。例如，在经营类游戏中，可以提供几百个不同的新游戏单元以供用户下载，每一个游戏单元带有数据（新图形、动画或者声音），同时还有隐藏在抽象接口之后的新代码。最初的可执行程序会把这些新游戏单元作为已有的组成部分进行对待。

游戏一般通过游戏对象工厂创建游戏对象。传递的是游戏对象的类型，函数返回的是某正确类型的游戏实体。

```
GameObjectType objType = LoadObjectType();
GameEntity * pEntity = factory.CreateObject(objectType);
```

对象工厂了解不同的对象类型以及相应需要的实例化类。这看起来似乎是采用以下的方式：

```
GameEntity * GameObjectFactory::CreateObject(GameObjectType type)
{
    switch(type)
    {
    case GAMEOBJECT_CAMERA:      return new GameCamera;
    case GAMEOBJECT_ENEMY:       return new GameEnemy;
    case GAMEOBJECT_TERRAIN:     return new GameTerrain;
    case GAMEOBJECT_PROJECTILE:  return new GameProjectile;
    }
    return NULL;
}
```

游戏在发售以后进行扩展的关键因素，是避免将这些对象类型固化到游戏对象工厂里。相反，对象工厂应该是可扩展的。最初，它不应该知道游戏对象类型和类的任何信息，而是通过代码的其他部分注册一个联系纽带，来完成特定对象类型和生成类之间的连接。

只要游戏探测到任何新的组件（第 11 章会详细讨论插件），就会加载这些组件并完成初始化。与这些组件需要初始化类似，需要做的第一件事情就是注册组件提供的新类和新的游戏对象类型。现在，当游戏试图装载使用到新游戏对象的新游戏关卡时，工厂会正确地创建这些对象，而游戏的其余部分会像对待任何其他通过抽象接口使用对象一样处理这些对象。

10.5　其 他 方 面

到现在为止，本章已经详细解释了使用抽象接口的好处。第 11 章将通过详细分析另外一个抽象接口应用——插件来继续对抽象接口进行讨论。然而，抽象接口不是针对任何问题的根本解决方法。尽管减少了实现和使用代码之间的耦合性，并在程序中总有一些事情激励我们去这样做，但是抽象接口本身也存在着一些问题。知道什么时候避免使用抽象接口和什么时候却要使用抽象接口同样重要。

第一个也是最主要的一个问题就是复杂性的增加。如果不需要抽象接口，那么使用它只是增加了另外一层复杂程度。它会使得程序更难理解、维护及将来的更改。在编程中，通常做想要做的任何事情的时候，最简单的方法就是最好的方法。

第二个问题与第一个问题紧密相关，使用抽象接口的代码调试困难。如果在程序运行过程中进入调试器，并试图查看通过抽象接口指针指向的对象元素的话，看不到任何内容。因为抽象接口没有实现，所以调试器不能显示出任何内容。为了看到这些内容，必须手工强制转换指针类型（这里假定知道指针指向的对象类型）。

抽象接口的最后一个不足就是它的性能。大家知道，依据接口的性质，在抽象接口中的每个函数必须声明为虚函数。这就意味着会在调用这些函数的时候出现微小的性能损失（这一点不是非常重要），另外它永远不能实现内联（这一点对于小函数来讲更为重要，参考第1章和第5章有关虚函数的性能介绍）。

最后一个问题就是抽象接口位于程序不同的层次可能会有不同的后果。需要考虑的最重要的方面就是在安排抽象接口的地方是否会使逻辑更合理，然后再考虑性能问题。如果抽象接口放置在很底层，那么一个框架会调用它很多次，就会导致性能降低。而把抽象接口提高到一个稍微高点的层次，就可以大大减少对接口的调用，同时还不影响整体性能。

还以渲染的例子讲，糟糕的设计就是使用抽象接口来处理单个的多边形。这样，在每个框架里就有几十万，甚至是数百万个多边形需要渲染。每次渲染一个多边形付出的成本是非常浪费的。相反，渲染器可以在诸如网格这样较高的几何层次处理，从而通过调用抽象接口一次对整个一组三角形实现渲染。

10.6 结　论

在这一章里讨论了抽象接口的概念及用途。首先学习了如何在C++中利用不带实现的类和纯虚函数来创建抽象接口。抽象接口的特定实现继承了抽象接口，并提供自身的实现。

然后讲述了抽象接口如何作为绝缘层来达到实现和使用接口代码之间的分离。这样就允许我们能更容易地切换类的实现，同时也提供额外的封装措施，从而使得文件之间的物理依赖较少。

接着考虑了抽象接口作为类的特征来执行，以及如何创建不依赖特定类类型的代码。当然，这些代码对已实现的抽象接口进行检查，如果发现就开始作用在这些接口上。这也是可以在游戏发售以后，不必发布新的可执行程序就创建新类的基础。

最后，讨论了抽象接口的一些缺点，以及最好在什么情况下避免使用或者谨慎使用抽象接口。

10.7 阅 读 建 议

以下是一本在介绍抽象接口概念方面不错的参考书目：

Llopis, Noel, "Programming with Abstract Interface", Game Programming Gems 2, Charles River Media, 2001.

这里有一本在COM内在机制方面很有启发性的读物，采用了循序渐进的讲述方法，从头开始开发了COM的API，并逐步给出了论证：

Rogerson,Dale, Inside COM, Microsoft Press, 1997.

下面的这本书仔细分析了组件软件，利用这种整体思想，可以把程序编写为独立的部分，然后通过抽象接口或者其他类似机制联系在一起。

Szyperski, Clemens,Component Software: Beyond Object-Oriented Programming , Addison-Wesley, 1999.

接下来的这本书在讲解许多其他基础模式的同时，讲述了工厂模式：

Gamma,Erich, et al. , Design Patterns, Addison-Wesley, 1995.

第11章 插 件

本章将讲讲：

- 对插件的需要
- 插件结构
- 插件的组装
- 插件的应用

本章将解释插件的概念，并讲解如何来构建使用插件的程序。读者可能会设计一些自己的工具，以便通过插件对其进行扩展，也可能只是想编写用于通用建模以及纹理管理的插件，以便在游戏开发中使用。无论哪种情况，本章会讲述如何实现插件以及如何组织使用插件的程序结构，读者会对插件有更深刻的理解。

11.1 对插件的需要

复杂工具不是一锤子买卖。人们为了满足自己的需要会去扩展这些工具，定制这些工具，或者添加新的功能。然而，把每个单独的小功能都添加到工具里不仅不太可能，而且也不是一个好办法。因为成千上万个不同的选项，会使得程序快速退化为不可维护的混乱状态，更别提程序在大小和特征方面的膨胀，以及会成为任何人都不乐意使用的、看起来庞大的、有点臃肿的软件了。

比较灵活的方法就是使用插件。程序提供了多数用户想使用的所有核心功能，而其余功能由称为插件的扩展部分提供。这样，就可以根据需要来选择使用哪个插件，如果不需要，甚至可以不装载相应的插件。另外，允许扩展工具也使得一些 API 成为公用的对外的接口函数，用户以及第三方公司都可以利用这些 API 来创建自己的插件，以此来扩充程序的功能。有时会注意到自己是在为别人的程序编写插件，然而有时又是在为了让别人可以修改自己的工具而编写着自己的代码。

11.1.1 用于其他程序的插件

公司里的建模人员非常兴奋，因为他们喜欢的 13.75 版本建模工具发布了。他们迫不及待地安装并执行了这个工具。这个软件包正是他们期待已久的，里面既有这个特征也有那个特征，还有另外的功能等，但没有那个功能。“但是我们在游戏里需要的，恰恰就是

那个功能！没有该功能我们就无法按时完成工作！”建模工作人员大声叫嚷着。可喜的是，这并不意味着必须让建模工具项目组修改该工具，你还有选择的余地。

一个可能的选择，就是使用另外一个建模工具来获得需要的所有特征。真的存在这样的程序吗？可能并不存在。另外一个选择是同时使用几个建模程序，这些工具加起来很可能具有期望的所有特征，这是很有可能的。但是建模工具都比较昂贵，那么费用的问题如何解决呢？还有就是互操作性怎么样呢？能够从所有这些工具里保存和加载模型吗？这些都是专家的问题。那么建模工作人员能够有效地使用这些工具吗？

第三个选择就是从头编写一个工具。这个工具可以替代建模工具，尽管基本不太可能拿出一个可以替代几百人的团队开发了十三代的建模工具，但是或许我们的工具能够对这些工具进行补充。建模工作人员可以依靠具有所有特征的工具来保存自己的模型，用我们的小工具装载游戏，并附加上想要的其他特征。来回地在程序间切换尽管有点讨厌，但是确有可能奏效。主要的问题是建模工作人员是否能够回到他们的建模程序，然后再对自己的工具已经处理过的模型继续进行操作。保证数据的双向有效性不是一件很容易的事情。当模型重新被存储到初始建模包中的时候，利用我们的工具附加上的信息会被覆盖吗？

最后一个可能就是扩展现有的程序，这个应该是理想的方法。建模工作人员想要的特征可能很小，因此要是可以扩展建模工具的话，那么这个方法应该是更好的解决方法了。要是程序使用的是插件结构，那么就会幸运好多，需要做的事情可能就是实现一些抽象接口，然后将插件交付给建模工作人员。所做的全部工作就是一个下午的事情，然而却能够成为建模工作人员心目中的英雄。

通常看起来，游戏公司一般会扩展市面上提供的建模程序，然后输出用自己的格式表示的模型和材质，用游戏特定的额外信息标记模型，对问题进行特殊的过滤和转换，甚至是在建模程序里用自己的游戏引擎显示模型的渲染情况。实际上，多数程序（不仅仅是建模工具程序）倾向于通过插件——网络服务、浏览器、MP3 播放器、E-mail 程序、编译器以及编辑器来实现功能的扩展。

11.1.2 为自己的程序编写插件

英雄地位是好的，并且可以为其他程序编写插件也确实是有用的，但是怎么对待自己的程序呢？为什么会想到不怕麻烦来增加插件呢？我们有自己的源代码，因此可以在需要的任何时候修改程序，然后再编译，最后发布带有新功能的游戏。

最简单的一个答案就是，这样的工具是最容易扩展的。一旦使用了插件结构，毫无疑问，最简单的方法就是为工具编写新的插件，而不是向其增加基本的功能。

要支持插件的使用，另外一个关键点就是最初的工具要与插件内容彻底隔离。这样就有可能具有了一个通用工具，并可以在不失一般性的情况下增加游戏相关的功能。例如，考虑一下资源查看器程序，利用查看器可以浏览游戏中使用的资源，例如纹理、模型、动画以及特效等，程序可能只是知道一些基本的、通用的资源类型，例如.tiff 格式的纹理、

VRML模型文件以及少数其他类型的资源。那么可以编写新的插件来浏览与自己游戏相关的资源：在自己平台上优化过的网格文件，特殊的压缩纹理，游戏实体对象等。

这样一个通用工具可以用于显示来自几个不同项目的资源，每个项目有自己的自定义资源类型。如果要在基本的工具上增加这些功能，就需要使工具依赖于这些项目以及未来可能的任何项目。更糟的是，如果两个不同的项目用略微不同的方式解释相同的资源会怎么样呢？而利用了插件，就可以选择自己想要显示的工具。同样的方式也适用于跨平台项目。可以利用编写的新插件，甚至根本不需要接触初始工具，就能够显示为特定平台创建的工具。

使用插件的最后一个优点，那就是如果将来有一天决定向第三方的游戏开发公司发布自己的插件的话，那么其他公司就可以为我们的游戏创建新的内容。从这个角度讲，其他人可能也想通过与扩展已有建模工具一样的方式扩展我们自己的工具，那么他们是否也可以有机会获得英雄地位呢？

11.2 插件结构

本节要对怎样建立一个插件结构进行详细论述，另外，还会讲述实现过程中的一些非常有趣的部分。随CD-ROM发布的源码，包括全部的Win32样例程序，可以在文件夹\Chapter 11.Plug-ins\下找到这些源码，同时 Visual Studio 的项目文件是 pluginsample.dsw。可以参考这些代码来获得更详细的情况。

11.2.1 IPlugin

所有的插件结构会围绕着为插件提供的抽象接口进行组织（参看第 10 章关于抽象接口的详细解释）。抽象接口包含了程序中可以使用的操纵插件的所有函数。

给出一个例子，为一组插件创建一个接口以便从工具中导出数据。以下是为插件提供的一个可能抽象接口：

```
class IPluginExporter
{
public:
    virtual ~IPluginExporter(){};
    virtual bool Export(Data *pRoot)=0;
    virtual void About() = 0;
};
```

以上的接口只声明了与所有输出方插件交互中需要的最少函数集。这个例子中，程序只可以用两种方式与插件进行交互。第一种方式是通过调用 Export 函数并传递合适的数据

来实现输出。每个插件可以用不同的方式实现这些函数，最后把相同的数据用不同的格式输出。

程序与插件交互的第二种方式是调用 About 函数。About 函数显示了插件的名称、构建日期以及版本等信息。在操作插件的时候，有类似这样的函数是非常有用的。它不仅允许我们查看当前加载的是什么插件，而且可以检查插件是所需要的正确版本。既然插件可以彼此独立并在脱离主程序情况下进行复制和更新操作，那么能够从程序内部检查版本就会消除用户的好多麻烦。

在这个例子里，假定在构造函数里进行所有的初始化工作，而在析构函数中执行所有的结束工作，因此这里看不到单独的初始化函数和结束函数。在可能的情况下，因为一个对象在没有初始化前是不可以成功创建的，所以这是进行初始化和结束操作的首选方式。另一方面，如果初始化失败了（例如，如果初始化必须访问一个文件或者分配内存），那么就需要从构造函数抛出异常（参见第 5 章异常处理）。

11.2.2 创建特定插件

下面创建一个导出指定格式的插件，这么做的目的只是做一个简单的练习。这个例子的代码如下：

```
class PluginExporterHTML :
{
public:
   PluginExporterHTML (PluginMgr & mgr);

   //IPlugin interface functions
   virtual bool Export( Data * pRoot);
   virtual void About();
private:
   //Any functions specific to this implementation
   bool CreateHTMLFile();
   void ParseData(Data * pRoot);
   //...
};
```

与上面给出的头文件一起，编写了一个.cpp 文件来实现这些函数，这样，编写的导出插件就可以使用了。在这里不需要做任何其他事情。需要注意的是不必更改原始程序里的任何一行。

创建另外一个可以输出不同格式的插件只需要生成一个从 IPlugin 继承过来的新类，然后用新的实现添加到空白地方就可以了。

```
class PluginExporterXML
{
public:
   PluginExporterXML(PluginMgr & mgr);

   //IPlugin interface functions
   virtual bool Export (Data * pRoot);
   virtual void About;
private:
   //Any functions specific to this implementation
   //...
};
```

11.2.3 处理多种类型的插件

到现在为止，已经看到如何来创建一个相同类型的新插件，在前面的例子里，插件都是导出类型的。通常一个程序不只需要一种类型的插件：一种导出数据的插件，一种从其他地方导入数据的插件，显示新数据类型的插件，或者是用于用户扩展的其他插件。可能性是多种多样的，因此，在较复杂的程序里显然需要多种类型的插件。

对多种类型的插件进行组织，最好的方法就是使用继承。一个抽象接口将包含对所有插件类型都适用的通用接口函数，例如初始化函数和停止函数（假定用户对这些任务有显式函数），得到插件名称、版本以及其他相关信息的函数。在例子中适用于所有插件的基本抽象接口代码如下：

```
class IPlugin
{
public:
   virtual ~ IPlugin(){};

   virtual const std::string & GetName() = 0;
   virtual const VersionInfo & GetVersion() = 0;
   virtual void About() = 0;
};
```

对于每一种插件类型，将创建一个从 IPlugin 导出的新类，在新类里添加插件类型对象的新功能。需要注意的是，因为这些新类是从抽象接口继承下来的，所以仍然是抽象接口，它们没有提供任何实现。以下是用于不同插件类型的类：

```
class IPluginExporter : public IPlugin
{
public:
   virtual ~IPluginExporter(){};
   virtual bool Export ( Data * pRoot) = 0;
};

class IPluginImporter : public IPlugin
{
public:
   virtual ~IPluginImporter(){};
   virtual bool Import(Data * pRoot) = 0;
};

class IPluginDataViewer : public IPlugin
{
public:
   virtual ~ IPluginDataViewer(){};
   virtual bool Preprocess ( Data * pData)=0;
   virtual bool View ( Data * pData, HWND hwnd) = 0;
};
```

要创建一个特定的插件，就需要继承指定类型的插件，因此除了需要实现通用 IPlugin 的函数，还需要实现特定于某类插件的函数。最终的继承关系树如图 11.1 所示。

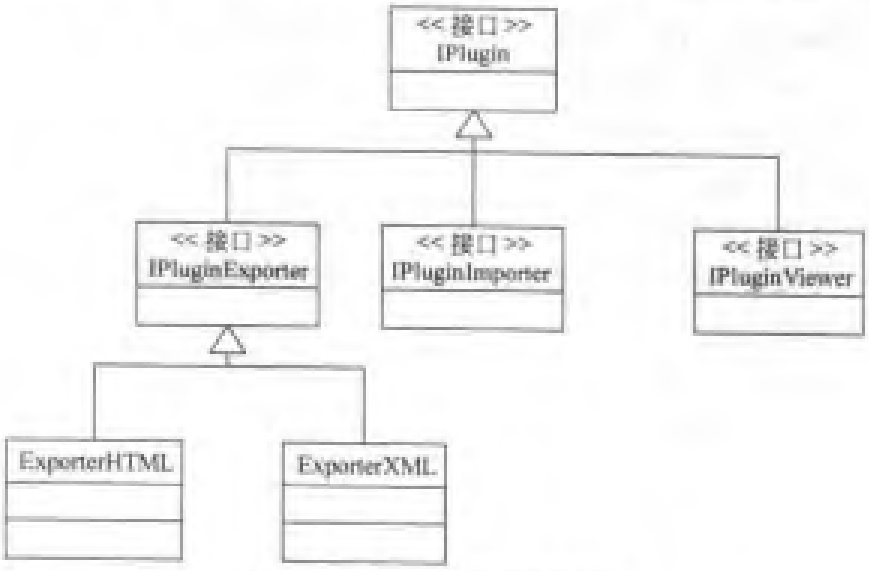


图 11.1 多种插件类型的继承树

11.2.4 装载插件

迄今为止，一直在讨论插件，描述插件接口和实现，但是，对于在运行时是如何装载这些插件并没有提到。谈到运行期装载插件，很显然它与没有使用插件的程序没有关系。否则的话，一旦程序进行了编译，就不能添加任何新的插件了。替换的做法应该是对插件进行单独编译与动态加载。

在 Win32 例子里，运行中加载插件最直接的方法就是使用 DLLs（动态库）。DLLs 允许将与库中代码的连接推迟到运行时刻。当显式地装载一个动态库时，动态库中的代码就可以提供给程序的其余部分，这看起来正好与我们的需求完美吻合。

要创建一个动态库，就必须指定在编译器里进行特定设置。利用设置指定目标是动态连接库（.dll）而不是可执行程序（.exe）或者静态库（.lib）。在编译器的支持下，用户也能在向导指引下，仅通过鼠标的单击建立一个 DLL 项目。

一旦创建了一个 DLL，就必须决定输出什么函数。不像其他类型的库，DLL 要求显式地标记这些向库外提供的函数、类或者变量。为了使事情尽可能简单，希望 DLL 的输出数量最少。输出的不是插件类本身，而是一个全局工厂函数，这个函数负责创建实际的插件对象。以下的代码实现工厂函数的输出：

```
#define PLUGINDECL __declspec(dllexport)
extern "C" PLUGINDECL IPlugin * CreatePlugin( PluginManager & mgr);
```

在 Visual C++里，从 DLL 中输出函数的方法就是在函数声明前面添加关键字——declspec(dllexport)，在上面的例子里就是 PLUGINDECL 定义的部分。下面将看到，为了使得 DLL 在函数输入的时候仍然可以重用相同的头文件，不是直接在代码里添加——declspec(dllexport)，而是定义了 PLUGINDECL。

对于工厂函数的声明，另外一点看起来似乎比较古怪的地方就是 extern "C"的使用。这个关键字告知编译器不要用通用 C++的方式来修饰函数名称，相反地，把该函数名称看作是一个 C 函数，不要对名字进行修饰。这样就可以使用函数的正常名字而不是编译器形成的名字来搜索使用的函数。

现在，除实际的插件实现之外，每个插件 DLL 需要提供工厂函数的一个实现。多数情况下，这个函数因为只创建一个正确类型的新插件，所以显得很简单。

```
PLUGINDECL IPlugin * CreatePlugin(PluginManager & mgr)
{
  return new PluginExporterHTML(mgr);
}
```

DLL将作为插件进行使用了。在有许多不同类型插件的工具中（或者是即使只有一种类型插件的工具，只为了增加安全性）经常使用的一个方便窍门就是对文件重命名，不使用通用的.dll扩展文件，而是改变成一个反映插件类型的后缀。例如，可以给导出插件赋予后缀.exp，为导入插件赋予后缀.imp。在有多种不同类型插件情形下，当试图手工管理这些插件时，这个小小的窍门就显得非常有用。

11.2.5 插件管理器

到现在为止，已经详细讨论了如何创建一个插件，但是没有涉及到如何使用插件这个问题。一直假定程序会自动地装载和使用插件。实际上这属于插件管理器的工作。

插件管理器的惟一作用就是管理插件。通过插件管理器，可以把任何事情都变得非常直观，但是却需要做大量的工作。插件管理器的目的是让程序的其余部分将插件作为对象使用，还并不需知道插件是动态装载进来的。

对于使用插件的程序而言，需要在插件真正要被装载的时候做出决定。在用户决定装载的时候，插件可以在程序开始时就装载，也可以是选择单一的插件进行装载。与CD-ROM（\Chapter11.Plugins\pluginSampLe.dsw）中给出的例子类似，多数程序尝试在程序开始的时候装载所有插件。要改变装载时间也很简单，在不同时机调用插件管理器的正确函数即可。

在我们的例子里，只要程序一加载，插件管理器就试图装载所有可用的插件。查找一个指定的目录，然后找到所有的文件，这些文件的文件后缀与需要处理的插件类型相匹配。因为用户可能想直接增加、删除插件，所以目录通常要与可执行程序有关联，并且目录要显而易见。在我们的例子里，可以在./plugins/目录里查看插件。

对于发现的每一种可能存在的插件文件，我们会把它作为插件装载，第一步就是要装载动态库：

```
HMODULE hDll = ::LoadLibrary(filename.c_str());
```

Win32函数LoadLibrary会根据它的文件名装载一个动态库，同时返回一个句柄，这个句柄在以后释放的时候是要使用的。如果动态库加载失败，那么函数会返回空NULL。因为插件文件有可能破坏了或者是偶然有与插件文件后缀相同的文件存在，所以在进行每一步骤时，都必须检查执行结果成功与否。

还记得在IPlugin头文件里是如何声明CreatePlugin函数的吗？这里再次给出这个声明，这次包括了所有细节：

```
#ifdef PLUGIN_EXPORTS
   #define PLUGINDECL __declspec(dllexport)
#else
   #define PLUGINDECL __declspec(dllimport)
```

```
#endif

extern"C" PLUGINDECL IPlugin * CreatePlugin( PluginManager & mgr);
```

现在可以清楚地看出，根据 PLUGIN_EXPORTS 可以把 PLUGINDECL 声明为两种不同的类型。插件管理器和插件自身的实现都需要包含 Plugin.h 头文件，但是插件实现还需要把它的函数声明为 DLL 输出，而插件管理器需要声明为 DLL 输入。在预处理程序上施加小小技巧，就能够只使用一个头文件，避免了在几乎相同的代码里维护两个独立头文件的麻烦。假定 DLL 已经正确加载进来，接下来的一步就是必须寻找输出工厂函数。

```
CREATEPLUGIN pFunc = (CREATEPLUGIN) :: GetProcAddress(hDll ,
                                        _T("CreatePlugin"));
```

GetProcAddress 函数从特定 DLL 的输出函数列表里查找指定函数。我们决定对所有不同的插件调用工厂函数 CreatePlugin，所以就只查找这一个函数。注意，可以利用普通名字进行函数查找，原因是把函数选择输出为 extern "C"，这样就告诉了编译器不在符号表里应用通常的 C++修饰。

假定已经找到了函数，GetProcAddress 将返回一个指向这个函数的指针，否则会返回 NULL，这样就知道没有得到想要的 DLL。

插件管理器最后准备创建插件了。通过调用刚得到的函数指针来调用工厂函数，这样就可以得到一个正确类型的插件。当然，插件管理器对插件的类型一无所知。这些是隐藏在 IPlugin 抽象接口后面的。

```
IPlugin * pPlugin = pFunc(*this);
```

在这里，因为工厂函数有一个插件管理器的引用参数，所以把*this 作为一个参数传递给工厂函数。为了与程序的其余部分交互，插件需要访问管理器，这些会在下一节里介绍。最后，插件管理器把刚创建的插件的指针放到列表中，以供在需要的时候由程序进行访问。

插件管理器的其他功能就很直接了。我们需要为程序提供一些枚举插件的方式。在给出的例子里，只给出了一个可以返回装载插件数量的函数，以及另外一个返回值为对应一特定索引的插件的函数。如果有多种类型的插件的话，事情就有点复杂了，这时就需要程序查询所有的导出插件、导入插件等。

这里也需要提供函数来实现运行期插件的装载和卸载。至少在程序退出的时候，为了可以正确释放内存，我们想卸载所有的插件。要释放一个插件，所需要做的全部工作就是从插件列表里移除插件，删除实际的对象，并通过调用 FreeLibrary 卸载相应动态库。卸载动态库 DLL 的函数有一个参数就是 LoadLibrary 返回的 DLL 句柄，因此必须确保保留了那个句柄和插件指针。

另外一个非常有用的特征就是，在不必停止程序的前提下，可以重新装载所有的插件（或者只是特定的某一个）。尤其在开发阶段，这种方式就很有用，如果装载主程序和数

据集需要一点时间的话，那么利用这个方式就可以快速对插件进行循环测试（指无须关闭主程序的情况）。为了能够正确地实现这个功能，就不只是需要调用一个函数来重新加载所有的插件了，必须能够单独释放，然后重新装载插件。这样做的原因就是只要动态库装载了（通过调用函数 LoadLibrary），那么它的文件就会被操作系统锁定，这意味着无法用新编译的新版插件来替换原来的插件。因此必须首先卸载插件，更新动态库，然后再重新装载。总而言之，这样做通常还是比强迫关闭程序然后再重新启动程序要快许多。

11.2.6 双向通信

到目前为止，所有在程序和插件之间进行的通信都是单向的，程序在需要的时候调用插件的函数。是导出文件的时候吗？调用合适的插件里的函数。需要显示一个带有插件的对象吗？调用渲染函数。一切都正常地进行着。

有这样的情况，有时候希望插件充当更积极的角色，例如向工具条增加按钮或者向主菜单添加一项。可能插件需要周期性地检查一些事情，或者需要在程序得到它以前截取一个消息。为了可以实现这些功能，就必须把程序结构组织好，以便允许插件进行相应的访问操作。

要保证插件对程序其他部分的访问，最清晰直观的方法就是通过插件管理器进行。因为在插件构造时，把插件管理器作为一个参数传递给了插件构造器，所以所有的插件都知道插件管理器。这样就可以比较容易地把插件管理器扩展为访问程序其他部分的通路。

接下来应该决定计划让插件具有什么级别的访问能力。对程序其他部分的访问限制越多，插件和程序之间的依赖性就越小（这意味着插件可能与未更新的程序版本一起工作），另外也意味着插件所能做的事情也就越少。给予插件自由支配的权利，不仅意味着插件可以做编写者决定让做的任何事情，而且意味着插件与程序的其余部分有比较紧密的耦合关系，还意味着更容易出错。可以遵循的一个比较好的原则就是：在允许插件完成需要功能操作前提下，尽量对其施加更多的限制条件。

最为安全和严格的方式，是通过插件管理器来完成任何操作。插件管理器需要预计出所有插件的需要，并为插件想要的每个操作提供函数。因为插件管理器是和程序的其他部分一起编译的，所以它对程序的不同部分有更多直接的了解，这样，通过插件管理器访问程序要比通过插件直接访问程序更不易出现问题。另外，随着程序改变以及新版本的出现，插件管理器也可以进行必要的改变。如果插件直接访问程序的话，一旦有大的结构上的变化，这些插件就是无效的了，必须更新甚至重新编译，这种情况实际上是要尽量避免的。

在 CD-ROM 的样例代码中（\Chapter11.Plugins\pluginsample.dsw），插件与程序的通信以最小限度的方式进行。因为是导出插件，所以与程序的其他部分没有太多交互。然而，样例程序在菜单里增加了一个入口 File|Export 子菜单。为了实现这些，程序使用了一些插件管理器里的函数来从输出子菜单中增加和移除条目。这种情况下，就需要插件管理器发现主框架的句柄，获得菜单，找到正确的子菜单然后插入菜单。如果把这些操作留给插件，

那么因为这些操作与程序结构有很紧密的联系，所以存在许多潜在的问题。

另外一个办法是赋予插件完全访问程序其他部分的权利。插件管理器返回一个主视窗的句柄，或者是指向主对象的一个指针，然后插件可以实现自由操作。因为这样允许最大的实现自由度，同时可能会做一些初始设计中没有预见到的事情，因而这是一个很危险的方法。

两个方法可以结合使用。插件管理器可以提供许多安全函数，给插件提供需要时访问程序的“后门”。多数插件将工作得很好，而且使用的也是安全函数。如果有人完全需要做其他事情，那么他们可以这么做，只是插件有可能在以后的运行中突然出现问题，这就是代价。

11.3 插件的组装

CD-ROM 上的源代码包含了一个基于插件的应用方面的例子。在\Chapter11.Plugins\文件夹下，包含了 Visual Studio 的 3 个不同项目文件。

- pluginsample.dsp：这是主要的 MFC 应用。包括插件管理器类。除了作为一个内核来说明插件是如何连接到一起的之外，它自身不做任何事情。
- pluginA\pluginA.dsp：这是一个简单的导出插件。为了简单起见，只是在输出函数调用的时候显示一个消息框，而不是输出内容。这里也有一个“About”对话框，通过这个对话框可以在应用里查看已加载的插件。
- pluginB\pluginB.dsp：另外一个简单的导出插件。

插件应该在与应用相关的./plugins/目录下，并且是在应用开始的时候加载进来。可以卸载所有的插件，并通过菜单全部装载。

11.4 插件的应用

已经详细地介绍了如何建立一个插件结构，但是插件如何在实际中应用呢？

11.4.1 使用插件

插件做起来这么复杂，值得吗？回答是“值得”。插件是扩展程序的主要方法，并且许多商用软件也设计为这样的方式进行扩展。有些产品甚至走向极端，把所有的功能都通过插件来实现，结果程序本身就是一个空壳，由插件来完成所有有用的工作。

对工具而言，插件确实是不错的，但是对于游戏究竟怎么样呢？在这里，通过插件来扩展功能一般不常用。许多游戏是可以由用户扩展的，但是扩展的方式通常是以新资源或者新脚本的方式，而不是新代码方式。提供完全基于插件方式的扩展，其主要问题之一就

是安全性。一旦插件连接到一起，那么事实上就可以操纵游戏甲的任何事情，甚至是操纵整个计算机。这就有可能让恶意代码做出许多不愉快的事情，从窃取密码和私人信息到从硬盘上删除文件。另外一方面，脚本代码通常在操作类型上有非常严格的限制。因此，与运行插件相比，运行不明渠道的游戏脚本要安全得多。

大概调试插件首先是一件讨厌而困难的事情，但是在现实情况下，却变得异常简单。大多调试器允许设置断点，允许像使用可执行程序一样，从已加载动态库中跟踪进入源代码，这样，只需要在设置断点以前等待 DLL 的装载就行了。另外，因为可以在不需要关闭主程序的情况下重新装载插件，所以就使得调试插件相对轻松愉快多了。

11.4.2 不足

然而，在插件的世界里不总是玫瑰色的。因为必须经过插件管理器（或者得到某个对象，或者得到对象的句柄，然后以其他的方式访问程序），所以从插件到主程序的交互总是稍微有点麻烦。

根据插件的动态本质，插件在处理全局数据方面有些困难。那么，既然无论如何这个都不是好的习惯，那么去掉全局变量，而是提供 singletons（单件模式）或者对象来访问这些全局数据，则多数问题可以得到解决。按照这样的思路，让插件使用 GUI 资源，有时就得讲究一点技巧了。同时，必须跳出一些限制，以保证所有工作如预期的那样进行。

另外，插件之间的交互尤其糟糕。两个彼此依赖的插件可能不是立刻同时加载的，或者它们加载的次序与希望的次序不同。通常，最安全的方法就是确保插件之间不存在彼此依赖关系。

除了这些不足，插件的另外一个潜在缺点正是来自于插件试图解决的问题。因为插件的本意是在编译以后可以扩展程序，所以必须直接安装、卸载插件，或者通过一些安装程序进行。这样的话，插件就有可能过期或者出现版本不匹配。可喜的是，插件通常是只为一个程序编写的，因而至少不会因为不同的程序版本安装以后，形成同一目录下的插件版本冲突。这种冲突能够导致 DLL 的崩溃，有时这会出现在共享动态库里（指几个不同的软件都使用同一个 DLL）。

谨慎对待插件的版本是一个好的习惯。始终确保用户可以核对插件的版本号，并且要经常检查程序版本的正确性以及与之交互的其他插件的版本。

11.4.3 平台

本章主要精力专注于在 Win32 平台上运行的插件。如果不是为 Win32 开发会如何呢？即使目标平台不是 Win32，通常的开发工具也是运行在 Win32 平台上，因此本章讲述的所有内容都是有关的。Linux 和 Apple 操作系统有类似的装载动态库的机制，因此只需要改变平台相关的细节即可。

游戏终端（游戏机）的情况又怎么样呢？有些游戏终端也有 DLL 支持，但是有些却没有。如果正在一个不支持动态库的工作平台上工作，则必须多做一点工作来获得相同的结果。可能会使用段装载（segment loading，一种允许装载和卸载特定代码段的技术）以及指针调整（pointer fixup）来实现动态装载代码。这里真正的好处是只导出了工厂函数，它只需要加载和运行一次，而且还避免了导出全部插件类的复杂性。在实际的插件对象通过工厂创建以后，就可以几乎像常规对象那样来使用。

11.5 结　论

这一章讲述了非常有用的插件技术。插件允许在不必调整初始源代码，不必再重新编译任何东西的情况下扩展程序的功能。也可以使用插件来扩展他人的程序，例如市面上已有的工具。

我们看到了如何构建支持插件使用的程序，如何组织不同的插件，如何管理插件以及如何在 Windows 平台上加载它们。如果使用了其他方法来动态装载代码，那么插件还可能用在不支持 DLL 的其他平台上。这些技术在插件管理器的样例代码中得到了应用，这些例子程序可以参见 CD-ROM（\Chapter11.Plugins\pluginsample.dsw）。

最后，讨论了如何在实际中使用插件，以及插件存在的优点和不足。

11.6 阅 读 建 议

令人惊讶的是，对于这么一项有用的技术，却找不到太多关于这方面的出版读物。一些有关插件结构方面的描述，主要是从通过插件进行扩展的软件包的文档里获得的，例如 Discreet 的 3D Studio Max、Adobe Photoshop 以及其他软件。通过阅读这些包可以发现许多关于插件实际应用方面的有趣细节。

以下的参考内容是关于任何 Windows 系统编程方面非常好的参考资料。这里还包括 DLL 方面的一些很好的章节。

Richter, Jeffrey, Advanced Windows, 3rd ed., Microsoft Press, 1997.

第 12 章　运行期类型信息

本章将讲述：

- 不使用 RTTI 进行工作
- 使用 RTTI
- 标准 C++RTTI
- 自定义 RTTI 系统

随着游戏大小和游戏复杂性的增加，在游戏中使用的类的数量也显著增加。可能需要在成千上万个对象（这些对象分别属于上百个不同的类）之间实现通信。有时需要在这些混乱状态中间得到所处理对象身份方面的更多信息，例如对象所属类的名称，或者它们是否继承了特定的类。为了实现这样的目的，就需要有运行期类型信息（RTTI）的支持。

在这一章，将学习 C++中提供的 RTTI 系统，以及它的用途和性能开销。将会明白为什么有些时候使用自己的 RTTI 系统也是明智的，这里还会提供一个很简单，但是又非常强大的自定义 RTTI 系统，该系统可以直接在自己的代码中进行使用。

12.1　不使用 RTTI 进行工作

在以面向对象方式设计的游戏里，所有游戏对象之间以及对象和游戏者之间应该互相交互，还应该使用封装和多态。一切都在一个相对抽象的层次上发生，实际上，从来不必担心正在处理的对象属于什么类型。

上面说的属于理想情形。与对象在越高的层次上交互，对象之间的交互层次也就越高，对代码之间的依赖关系引入得就越少。现在考虑图 12.1 所示的实时策略游戏（RTS）中使用的简化层次结构。

对象之间的多数交互应该通过 GameEntity 类进行。换句话说，UnitTank 不应该对 UnitResourceCollector 有任何直接了解。但是，当坦克碾过小小的资源收集机器人（UnitResourceCollector）时会发生什么情况，我们是知道的。那么，如何才能不用判断坦克是否将要碾过 UnitResourceCollector，就可以实现呢？坦克和资源收集器都得到一个消息，消息指明它们正和另外一个 GameEntity 对象相撞，而不知道对方的特定类型。相撞的双方都可以得到与自己碰撞的 GameEntity 对象的速度和质量，并且可以决定是否有损坏的危险。在资源收集小机器人的例子里，这个危险是足够把它碾平的。

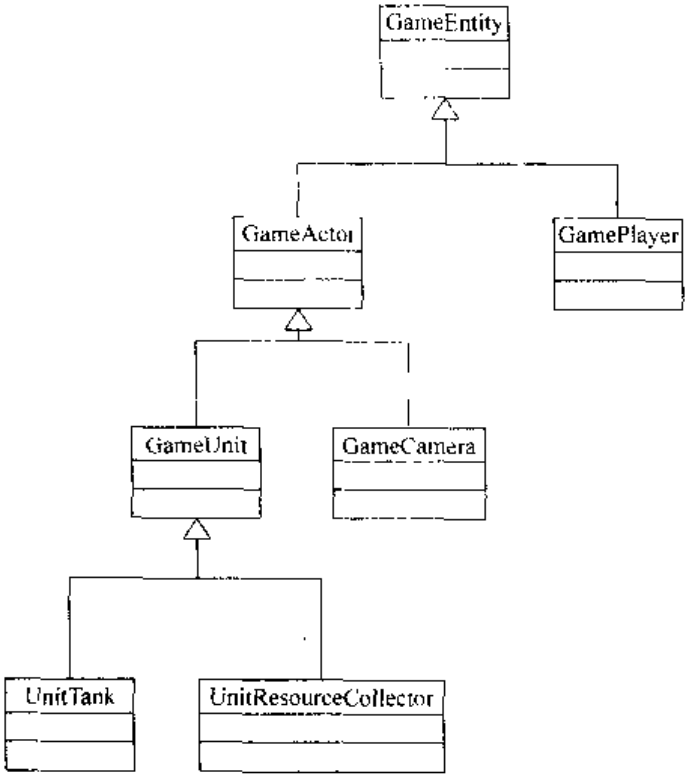


图 12.1　游戏实体的一个可能继承关系

这样的技术还可以用于其他类型的交互。例如，假定游戏里有装满高度易燃液体的桶。当一辆坦克撞上一只桶以后，桶检测到这个碰撞后会引起爆炸发生，爆炸会破坏位于一定半径之内的所有游戏实体。如果爆炸引起的破坏足够摧毁坦克的话，坦克就会爆炸——所有这些，都是在坦克不知道它自己撞到的是一个有爆炸危险性的桶的情况下发生的。同样，桶也不知道什么撞了它，只知道碰撞自己的对象足够引起桶发生爆炸。

用这样的方式对对象交互进行建模的主要优点是，在编码中不需要在所有对象类型之间引入太多依赖关系。另外，对象之间的相互影响是非常通用的，所以新的交互一定可以按照建模指定的方式自动进行。例如，现在不需要对坦克与汽车的碰撞建模做任何特殊处理，处理碰撞和毁坏的通用方法就会处理这些问题。因为地雷与易爆炸性的桶非常类似，所以如果需要使用地雷，那么也就变得非常容易了。只要增加了新的单元类型，就要实现一些通用的与外界进行交互的方法，那么利用这些方法就可以与所有其他单元很好地交互了。

12.2　使用 RTTI

考虑以下的替代方案：在游戏中，坦克类明确地进行检查，检查是否与其他对象类型存在交互。如果这样的话，坦克就需要对游戏中的桶、资源收集机器人以及所有其他单元有所了解。只要坦克遭到了碰撞，就需要找出撞击自己的是什么对象，然后根据对象类型

做出反应。每当添加了新的对象类型之后，就需要使其他所有已存在的对象知道新对象的加入，然后还必须手工编写已有对象与新对象之间的交互代码。这些看起来好像是一些不必要的工作，不过有的时候，却是非常方便，甚至是必需的工作，可以确切地知道正在处理的对象类型，然后在继承结构里执行强制转换，最后再直接使用对象。

在前面的实时战略游戏里，假定坦克单元具有与飞行器单元结合从而形成更强大单元的能力。要合并这些单元，需要把飞行器先着陆，然后再将两个单元集合到一起。如果只是利用通用代码来处理这种合并，那么双方就可能在侦测到碰撞的时候发生爆炸。替换的方法是，应该在坦克或者飞行器的碰撞处理函数里有专门代码来检查碰到的物体是不是指定的单元，如果是的话，就变形为新的单元。在类似这样的情况下，找出撞击对象的确切类型是非常有用的，但是这也只不过是一条捷径而已。

在更真实的情形里，可能想让其他几个单元与坦克一起，也具有单元合并生成新单元的能力。如果是这样的话，更好的设计应该是从 GameEntity 类中抽象提取出那些交互。在任意两个单元集合到一块的时候，单元之间发出消息，询问是否可以实现合并。如果可以合并，那么就触发合并的发生，否则忽略合并，然后按照正常情况处理执行。正如在第 10 章里讲到的那样，这样的功能或许利用抽象接口实现比较好。

从这个例子得到的启示就是，使用 RTTI 时应当保守谨慎。如果发现自己几乎想知道与自己交互的所有对象类型，那么通常情况下，这标志着该设计是一个不好的面向对象的设计。应该把 RTTI 技术认为是捷径而不是主要设计技术，因而需要谨慎地使用。过度依赖运行期类型鉴别会导致混乱的类依赖，使程序的维护、新特性的增加以及程序调试变得困难起来。

什么情况下使用 RTTI 才是一个不错的选择呢？对象和游戏者之间的交互可以从使用 RTTI 中受益。想免除创建新的抽象接口，或者向基类添加新功能的麻烦，在这样为数不多的场合下，大概可以使用 RTTI。合并两个单元就是一个很好的例子，只要它们是拥有额外功能的两个单元。在对一个场景进行渲染的时候，检查一个节点是否是灯光也是一个可以接受的例子。一旦对象之间的交互变得非常普遍了，就需要把这些功能移动到更高级别的层次上进行处理，并且要避免 RTTI 的使用。

也有这样的可能，对自己感兴趣的对象，无法调整对象的父类，这可能是因为使用的类来自没有源代码的库中，或者有源代码，但是由于与其他项目的兼容性问题，所以不能更改。既然那种类型的对象可能直接从库里创建，那么就无法强制它使用新类，因此为从那些类继承下来的对象添加功能就是不可能的了。在类似这样的情况下，使用 RTTI 来检测特定的类，并添加一些新的功能或者进行基于类型的对象间交互，可能就是惟一的方法了。

序列化是需要 RTTI 的另外一种情形（参见第 13 章对象的创建与管理）。序列化是保存和恢复一个对象的过程。可以在磁盘、内存甚至是通过网络来进行。在序列化过程中，要求一个对象向流中写入数据。写入流中数据的第一个信息就是记录将要保存的对象类型。类型可以是对象的类名称或者其他标识对象的惟一性的 ID。这实际上就是从 RTTI 系统里得到的信息类型。为了在以后或者是网络的其他终端恢复对象，首先查看保存在流中的对

象类型，创建一个那种类型的对象，然后读入这个对象的所有信息。

12.3 标准 C++ RTTI

如何正确地找出一个对象属于什么类呢？或者在更通用的意义上讲，怎么找出一个对象是否从一个特定的类继承过来的呢？要讲述的第一个方法就是使用标准 C++ RTTI 系统。然后介绍一个自定义的运行期类型信息系统。

12.3.1 dynamic_cast 运算符

在 C++ RTTI 系统中，最重要的部分就是 dynamic_cast 运算符。它的语法与其他任何 C++强制转换操作符相似：

```
newtype * newvar = dynamic_cast<newtype*>(oldvar);
```

dynamic_cast 的概念非常简单：只要对象的实际类型到想强制转换的转换类型合法，那么就会进行指针转换或者新类型的引用。如果 dynamic_cast 执行成功的话，就返回指向新类型的有效指针，如果想要进行的强制转换非法而导致转换失败，那么就会返回一个 NULL 指针。

强制转换合法意味着什么呢？合法的动态强制转换可以这么看待：如果想转换到的指针类型是对象本身的类型，或者是它的父类的话，都可以认为是正确的强制转换。这里指的对象父类不必非得是直接父类，任何祖先类都是对象的父类。

因此可以说，dynamic_cast 实际上做了两件事情。首先检查强制转换是否合法，接着真正地把指针转换到新的指针类型。多数时候，如果想知道一个指定对象是否是从特定类派生而来的，那么就还想通过刚才检查的那个类指针来控制那个对象，因此 dynamic_cast 仅用一步就完成这些操作，并免去了一些文字录入工作。这里有一些例子，展示 dynamic_cast 如何在图 12.1 的继承结构中使用。

```
GameCamera * pCamera = new GameCamera;
// This works fine
GameActor * pActor = dynamic_cast<GameActor*>(pCamera);
//this is also fine
GameEntity * pEntity = dynamic_cast<GameEntity*><pCamera);

//The next cast will fail because the object is of type
//GameCamera, which does not inherit from GamePlayer.
//The variable pPlayer will be NULL after the cast.
GamePlayer * pPlayer = dynamic_cast<GamePlayer * >(pCamera);
```

在上面的代码片断里，所有的转换都是从一个较低类向一个较高类转换的。这类转换因为通常是把指针向类层次结构的上方移动，所以常称为向上强制转换。

在大多数应用中，一种更常用的强制类型转换是向下强制转换。这种转换把指针向类层次结构的下方移动。在使用多态性并想知道特定对象更多相关信息的时候，这种转换尤其常用。

```
//pEnt is of type GameEntity,but points to a GameActor object
GameEntity * pEnt = new GameActor;

// This is fine
GameActor * pActor = dynamic_cast<GameActor * >(pEnt);
//Not OK. This is a downcast, but the cast is not
//legal because GameUnit is now an ancestor of
//GameActor, which is the type of the object we
// are manipulating.
GameUnit * pUnit = dynamic_cast<GameUnit>(pEnt);
```

到这里可能已经注意到了，dynamic_cast 表面上与 static_cast（参考第 3 章）非常相似。看起来它们都要对向上或者向下转换进行合法性检查，不同的是 static_cast 在编译时报告问题而不是在运行时返回空指针。

主要的不同就是 static_cast 对于强制转换的指针指向的对象的实际类型是不清楚的。而 dynamic_cast 却相反。只要是在一个相同的继承层次结构里，static_cast 就会执行任意转换：向上转换或者向下转换，它自身都不关心这些，都是可以接受的转换。而 dynamic_cast 将检查对象类型并只允许执行有效的强制转换。

除了在单一继承层次结构里的向上转换和向下转换之外，dynamic_cast 也可以在多重继承结构层次里执行强制转换，包括交叉强制转换（crosscasts）。现在考虑一下图 12.2 所示的继承结构关系。

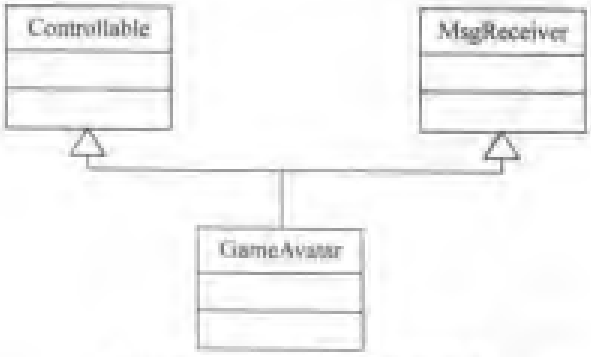


图 12.2　使用多重继承的类继承

给定一个指向 GameAvatar 对象的指针，可以正确地把它转换到向上及向下的任意分

支上：

```
//Going up
GameAvatar * pAvatar = new GameAvatar;
Controllable * pCont = dynamic_cast<Controllable*>(pAvatar);
MsgReceiver * pRec = dynamic_cast<MsgReceiver * >(pAvatar);

//Going down
Controllable * pCont = new GameAvatar;
GameAvatar * pAvatar = dynamic_cast<GameAvatar*>(pCont);
```

可能记得在多重继承章节已谈到过，多重继承进行向上和向下强制转换需要一些运行期的调整工作。为了确保总是访问正确类型的类，dynamic_cast 负责增加或减少指针的偏移量。也可以实现交叉转换，即在继承层次中做水平移动。

```
//Crosscast
Controllable * pCont = new GameAvatar;
MsgReceiver * pRec = dynamic_cast<MsgReceiver * >(pCont);
```

12.3.2 typeid 运算符

dynamic_cast 机制提供了特定对象继承关系方面的信息，并允许在不同类型之间进行强制转换。C++还提供了另外一个机制，可以支持直接找到有关对象的更多信息。

typeid 运算符返回一个有关对象类型方面的信息。重新回到前面例子里的类层次中，可以用如下的方式来使用 typeid：

```
Controllable * pCont = new GameAvatar;
const type_info & info = typeid(*pCont);
```

所有有关想要知道的对象的信息都包含在 typeid 返回的 type_info 中。在 type_info 里没有包含特别多的信息，其中最重要的信息就是类名称（以用户可读的方式和 C++加工后的版本两种方式提供）。

```
std::cout<<"Class name:"<<info.name();
//The output of running the above statement is Class name:GameAvatar
```

type_info 的另外一个用途就是用于类的类型比较。如果想确保两个对象确实属于同一类，那么就可以这么编写代码：

```
const type_info1 & info1 = typeid(*pObj1);
const type_info2 & info2 = typeid(*pObj2);
if(info1 == info2)
    //They are the same type . Do something..
```

注意正在进行的是对对象的 type_info 结构进行比较，而不只是比较对象的指针。对于属于相同类的对象来讲，会返回位于不同内存位置的相同 type_info 结构，因此必须比较对象的内容而不是保存对象的地址。

typeid 也不只可以用在对象上，还可以用在类型名称上。这样就允许通过比较两个 type_info 检查一个对象是否属于指定的类，其中一个 type_info 是从对象返回的，而另一个 type_info 是从想要比较的类返回来的。

```
if(typeid(*pEntity) == typeid(GameAvatar))
   //The Object * pEntity is of the class GameAvatar
```

在使用 typeid 时，需要记住的就是，它只能够有目的地应用于多态类型，也就是说，类中至少要有一个虚函数。如果在非多态类类型中使用它，那么返回的结果就是有关引用类型的信息，而不是有关对象的信息。因为不会不知道非多态对象属于什么对象，所以这个不是什么问题。

12.3.3　C++ RTTI 分析

标准 C++RTTI 系统对大多任务而言已经是足够用的了。它可以解决对象是否继承了特定类这样的问题，也支持在多重继承层次中的不同父类之间执行强制转换，并返回类的名称。对于所有的这些工作，都可以自动实现而无须做任何额外工作。那么为什么还需要做其他的事情呢？

标准 RTTI 系统的主要不足之一就是，如果编译器有关闭 RTTI 的开关的话（多数编译器都支持），那么 RTTI 就处于总是有效或者总是无效的状态。这意味着无论是否需要，带有虚函数的每个类（例如多态类）将会有与其相关的运行期类型信息。另外注意到，仅仅是带有虚函数的类（因此就有一个虚函数表 vtable）可以成为 RTTI 系统的一部分。因而所有的简单、轻量级类，例如点、向量以及颜色，仍会保持不变。

所有的多态类都具有与其相关的运行期类型信息，这样需要付出什么代价呢？情况不是很糟糕。需要记住，这些动态信息是针对每个类的，而不是针对每个对象的，因此不会增加对象的大小。通常，特定的细节信息要取决于每个编译器的具体实现，不过通常情况下，指向类信息的指针都保存在虚函数表 vtable 里，因此类信息本身要可以有足够的空间来容纳类的名称、少量指针、计数器及其相关信息。因此，所需代价一般在 30~50 字节的状态。如果偏高点估计，假设现在有 1000 个类（是类而不是对象），那么总计起来，也仅

仅是需要 50KB 的代价。与从典型的计算机以及游戏终端上可用的内存相比，这个代价是低廉的。但是对于只有有限内存的手持设备而言，代价就显得很大了。

在多态类型上执行 dynamic_cast 和 typeid 有什么限制呢？即使第一眼看上去感觉似乎限制很严格，但是实际上不是这么回事。在运行时，惟一需要获得的就是多态的类型信息，否则类型会在编译时就得到，那么就无须知道其动态信息了。

默认 C++RTTI 系统的最大缺点体现在性能方面：如果有的话，那么应该是因为缺少平台与编译器之间的一致性。在有些 RTTI 的实现中，dynamic_cast 会为了检查转换的有效性而做各种各样的验证，速度可能会慢一些。但是在其他的 RTTI 实现里，可能就相当地快速。性能也取决于正在被讨论的对象是否使用多重继承。

一个好的设计程序对 RTTI 的依赖应该是非常少的，潜在的性能开销可能不是任何问题。如果要在游戏执行过程中偶尔使用 RTTI 的话，那么其使用要限制在序列化操作上，如果真是这样的话，C++RTTI 系统就非常适合了。

不幸的是，这只是一个理想化的环境。因为遗留的原因或者只是因为游戏设计的原因，导致在运行期需要大量执行 RTTI 操作。如果每个游戏实体在每帧中都执行几个 RTTI 操作的话，一旦有 1000 个游戏实体，那么就会导致每帧中要执行好几万次 RTTI 操作。效率就会成为现在的关键所在，这种情况下，使用自定义 RTTI 系统显得很必要。

最后，另外一个使用自定义 RTTI 系统的原因，就是只想更好地控制 RTTI 系统。以后会看到，自定义 RTTI 系统不是特别的复杂，它是相当容易编写的。实际上，它只涉及很少的代码。因此，只要付出很少的努力，就可以保证在平台和编译器之间（编译器版本，不能低估在同一编译器的不同版本在实现方面的区别）操作的有效性，同时还可以清楚在性能方面的提升。

12.4 自定义 RTTI 系统

当决定使用一个自定义的 RTTI 系统时，确实必须问自己一下：通过它究竟想得到什么？如果只是复制一个标准 C++ RTTI，那么再编写自己的 RTTI 就没有什么意义了。现在用一个非常简单的方法开始，然后随着需求的增加，逐步建立起 RTTI。

12.4.1 最简单的方法

可能最简单的 RTTI 查询就是找出一个类的实际类型。如果这就是需要做的，那么这确实是非常容易做到的。如果是希望得到一个使用了所有最新 C++语言特征的对象，然后来自动获得该对象类型的话，那么你就会感到失望了。

1. 使用字符串

最简单的方法就是手工编写一个函数来返回类的名称：

```
class GameEntity
{
public:
   virtual const char * GetClassName() const
   {return "GameEntity";}
   //…Rest of the class goes here
};
```

这是非常简单的，只在感兴趣的类里写了一个函数，该函数返回了类的名称，这就是做的工作。实际上，在上面的代码中有一个小技巧和一个暗示的假定。小技巧就是函数要是虚函数。函数之所以需要声明为虚函数，因为查询的是对象的类名字，而不是指针的类名字。这些就是虚函数可以做到的，调用对象所属的类的函数，而不是指针所属的类的函数。

```
GameEntity * pEntity = new GameEntity;
pEntity->GetClassName();      //OK, returns "GameEntity"

GameActor * pActor = new GameActor;
pActor->GetClassName();       //OK, returns "GameActor"
GameEntity * pObj = new GameActor;
pObj->GetClassName();         //will only return "GameActor" if
                              //GetClassName is virtual!
```

隐含的假定就是返回类型的函数必须在所有类中都统一命名。如果不是这样的话，既然不知道对象是什么类型，而对象类型也正是最先想知道的，所以就不知道该调用什么函数。

要确保函数名称在任何地方都是相同的，为了节省输入工作，可以创建一个宏来自动为每个类增加那个函数：

```
#define RTTI(name) \
   public: \
         virtual const char * GetClassName() const \
           {return #name;}
```

宏的使用极其简单：

```
class GameEntity{
public:
```

```
    RTTI(GameEntity);
    //_Rest of the class
};
```

这个方法非常简单灵活。它可以适用于任何类，无论类中使用单继承还是多重继承。当然，这个方法还没有解决一个对象是否继承一个特定类这样的问题，在后面再详细讨论。

该方法的主要不足就是使用了特征字符串。为了利用函数 GetClassName()的返回结果做一些有用的事情，就必须使用字符串。字符串对于在调试器里打印和读取这样的事情确实是很好的，但是在比较方面就不是特别快，而往往需要做的就是比较。如果紧接着，开始一个大小写不敏感的字符比较，那么就会变得更慢。在过多依赖通过查找来得到对象类类型的程序里，就会很快意识到字符串操作带来的影响。

2. 使用常量

一个可以选择的方法就是用灵活性来换取性能，可以不返回字符串来标识，而是返回一个常量或者一个枚举。在只有很少数量的子类时，这就非常方便了。同时，如果不打算让程序其他部分对其进行扩展，就不需要担心在将来会创建新的常量或者生成新类型了，因此不存在常量冲突的问题。

```
Class GameUnit
{
public:
   enum UnitType
   {
         UNIT_PAWN,
         UNIT_RESOURCEGATHERER,
         UNIT_TANK,
         UNIT_PLANE
   };
   Virtual UnitType GetUnitType() const = 0;
};

class GameUnitPawn : public GameUnit
{
public:
   virtual UnitType GetUnitType() const
               {return GameUnit::UNIT_PAWN;}
};

class GameUnitTank: public GameUnit
{
```

```
public:
   virtual UnitType GetUnitType() const
                  {return GameUnit::UNIT_TANK;}
};
```

不是返回字符串作为类名字，而是选择采用了返回一个与游戏单元类型相关的特定常量。这种方法的主要优点体现在，验证一个单元是不是坦克会非常快，只需要比较两个整数即可。

该方法的一个主要不足体现在代码的清晰性方面。没有使用 dynamic_cast 和 typeid，也没有查询类名称。只是检查对象是游戏单元的什么类型，这个是在代码中想得到的真正内容。不要因为代码实现的简单，就低估它的能力。

3. 使用内存地址

可以合并前两种方法的优点，提出一个在检查对象类型方面既通用又快速的新的实现方法。为了实现这个方法，将使用惟一标识类的静态类变量的内存地址。变量本身将包含类的名字。这就提供了前面两个实现方法里具有的最佳特征。

```
class RTTI
{
public:
    RTTI(const string & name): m_classname(name) {};
    const string & GetClassName() const {return m_classname;}
private:
    string m_className;
};

class GameEntity
{
public:
   static const RTTI s_rtti;
   virtual const RTTI & RTTI() const {return s_rtti;}
   //…Rest of the class goes here
};
```

可以用与以前非常类似的方法使用它，只是现在浏览 RTTI 对象。如果保持一个字符串看起来有点麻烦的话，那么稍等一会，下节将讨论这一点，而且会把 RTTI 类变得更适用一些。

```
GameEntity * pObj = new GameActor;
pObj->RTTI.GetClassName();   //Will return "GameActor"
```

因为使用了类静态变量，就能够保证每个类只有一个地址，因此可以通过比较两个RTTI对象的地址实现对象的比较。

```
//Comparing pointers directly
if(&pObj1->GetRTTI() == &pObj2->GetRTTI())
   //They are of the same class…
if(&pObj1->GetRTTI() == &GameActor::s_RTTI)
   //pObj1 is of type GameActor
```

采用这样的方式工作以后，因为只是指针的比较，所以速度很快。但是必须得到RTTI对象的地址并直接比较它们就显得有些笨拙。如果把所有信息进行封装，然后只是比较两个RTTI对象是否相同，难道不是更好吗？

```
class RTTI
{
public:
   RTTI(const string & name) : s_ClassName(name){};
   const string & GetClassName() const;
   bool IsExactly ( const RTTI & rtti ) const
                {return ( this == & rtti);}
private:
   string s_ClassName;
};
```

既然已经把指针比较封装起来了，那么就可以以更简单的方式重写前面的代码。

```
//Comparing RTTI objects
if( pObj1-> GetRTTI().IsExactly( pObj2->GetRTTI() ) )
   // They are of the same class…
if(pObj1->GetRTTI().IsExactly(GameActor::rtti))
   // pObj1 is of type GameActor
```

如果对这样的做法感到迷惑，那么就为RTTI类提供一个operator==。这样就可以使用操作符而不是IsExactly()函数了。这只是个人偏好问题。使用operator==的一个好处就是使代码看起来更像是C++ RTTI系统，因此这可能是程序员已经使用的最熟悉的方法了。

可以更新宏来给类增加运行期类型信息。不幸的是，在用这个新的RTTI类的时候遇到了一个小障碍：需要提供一个类静态的声明和一个在.cpp文件中的定义。这意味着如果想使用宏的方便性，就需要创建两个宏，一个是用于头文件的，一个是用于.cpp文件的。

```
#define RTTI_DECL \
  public: \
        static const RTTI s_rtti; \
        virtual const RTTI & RTTI() const { return s_rtti;}

#define RTTI_IMPL(name) \
  const RTTI name::s_rtti(#name);
```

12.4.2　添加单继承

到现在为止，使用的简单的 RTTI 系统没有继承的概念，可以查询一个对象是否属于一个特定的类，但是不知道对象父类的任何信息。

如果把 RTTI 作为代码的一部分来使用，更为可靠的方式就是根据继承而不是根据特定类来考虑。例如，假定编写了一个函数，在每一帧里，这个函数遍历所有的游戏实体，然后把是镜头的实体放到一边，以便在以后的渲染中使用它们。

```
//Shaky code checking for exact class
GameEntityList::iterator it = entities.begin();
while(it !=entities.end()){
  GameEntity * pEnt  = * it;
  if(pEnt->GetRTTI().IsExactly(GameCamera::rtti))
        cameras.push_back(pEnt);
}
```

如果在以后的时间里，引入了从 GameCamera 继承下来的几个镜头类型会发生什么事情呢？代码将忽略它们。真正关心的是所有属于镜头的对象，或者是从 GameCamera 继承下来的对象，既然从 GameCamera 继承下来的对象都是一个 GameCamera 对象（参见第 1 章中的继承），因此，实际上希望按照以下的方式来编写代码：

```
//More robust approach using derivation
GameEntityList::iterator it = entities.begin();
while(it != entities.end()){
  GameEntity * pEnt = *it;
  if(pEnt->GetRTTI().DerivesFrom(GameCamera::rtti))
      cameras.push_back(pEnt);
}
```

现在可以扩展 RTTI 类，使其包含它的父类信息。通过增加指向父类 RTTI 静态对象的指针来实现这样的功能，这个指针在传递类名称的构造函数里被初始化。改进的 RTTI 类

如下所示：

```
class RTTI
{
public:
   RTTI(const string & className);
         m_className(className),m_pBaseRTTI(NULL){}
   RTTI(const string & className, const RTTI & baseRTTI):
         m_className(className),m_pBaseRTTI(&baseRTTI){}

   const string & GetClassName() const
         {return m_className;}
   bool IsExactly(const RTTI & rtti) const
         {return (this == & rtti);}
   bool DerivesFrom ( const RTTI& rtti) const;

private:
   const string m_className;
   const RTTI * m_pBaseRTTI;
};
```

做了新的修改之后，可以很容易地告知它的父类是谁了。但是前面的祖先是什么呢？通过父亲的父亲，以及它的父亲，直到找到了一个没有父亲的类为止，这样就得到了对象的祖先。如果按照这样的方式没有找到想要的类，那么就知道这个对象不是从这个类继承过来的。下面就是 DerivesFrom()函数的实现：

```
bool RTTI::DerivesFrom (const RTTI & rtti) const
{
   const RTTI & pCompare = this;
   while(pCompare != NULL)
   {
         if(pCompare == &rtti)
               return true;
         pCompare = pCompare->m_pBaseRTTI;
   }
   return false;
}
```

为了简单起见，可以把最新的变化封装到一个宏里。惟一的区别是需要两个不同的宏：一个用于没有父亲的类，而一个用于有父亲的类。实际上，只用一个宏已经做到了，如果

没有父亲的话，就传递 NULL。但是下面的代码显得更清楚一些：

```
#define RTTI_DECL \
  public: \
        virtual const RTTI & GetRTTI() { return rtti;} \
        static const RTTI rtti;

#define RTTI_IMPL_NOPARENT(name) \
  const RTTI name::rtti(#name);

#define RTTI_IMPL(name,parent) \
  const RTTI name::rtti(#name,parent::rtti);
```

最有意思的部分就是 RTTI_IMPL 宏。一眼看上去，似乎有点奇怪。把父 RTTI 静态对象作为参数。但是这些是在静态初始化时发生的，那么在调用游戏主函数之前，怎么知道父 RTTI 对象是如何被初始化的呢？毕竟，这里无法保证不同文件静态初始化的相对次序。答案应该是，我们不知道，另外也不关心。所做的全部事情就是保存内存地址以备使用，而不想从父 RTTI 对象读取数据或者调用函数，因为已经编译了程序，所以内存地址已经固定下来。因此，这种方法即使有点不太正统，却显得非常安全。

从这一点讲，具有 RTTI 信息的每个类也就知道其父亲和（潜在的）所有祖先了。虽然 RTTI 系统还非常简单，但是已经具有了所有的特征。惟一没有提到的就是多重继承。而如果我们使用单继承或者是带有 QueryInterface 函数的抽象接口，那么多重继承完全是不必要的。如果确实是这样的情况，那么这些已经足够在游戏中使用了。

如果需要多重继承支持，同时自己也乐意付出一点额外的内存和性能为代价的话，那么就还要阅读本章的下一节。支持单继承的 RTTI 系统源代码在 CD-ROM 中的 \Chapter1.2.RTTI\下（Visual Studio 工程是 rtti.dsw）。

应该留意的一点就是 DerivesFrom()函数的性能。所有其他的 RTTI 查询通常是返回一个指针或者进行简单的比较，所以速度都非常快。DerivesFrom()函数却有所不同，它需要循环到查找的类，或者到达特定继承树的根部才会停止。需要清楚的是，这个函数也应该是比较快的，因为并没有调用其他函数，而只是进行指针的比较，但是因为它要进行循环，所以没有其他函数快。

最差的情形就是想知道一个对象是否属于特定类，而特定类又并不是对象祖先的时候。这种情况下，RTTI 查询就需要遍历继承结构中的所有层次。如果有人把类层次设计为几十或者几百层的深度，那么性能方面的影响就非常显著了。当然，要是在这种情况下的话，与简单循环的性能问题相比，这里会有更大的问题来考虑了。还有就是要记住，这个函数不应该滥用。

12.4.3 增加多重继承

为了增加多重继承支持，仅仅扩展单继承的实现，使其可以继承多个父类。惟一需要做的改变就是 DerivesFrom()类，把该类调整为反复找到所有的父类，而不是像单一继承那样只找到第一个父类。

还是一步一步往下看吧。现在重新看一下第 2 章用到的 AI 多重继承类层次。这个特定的层次结构更适合使用 RTTI 方法，而图 12.2 中的层次结构最好是用抽象接口和 QueryInterface()方法实现。AI 的继承层次如图 12.3 所示。

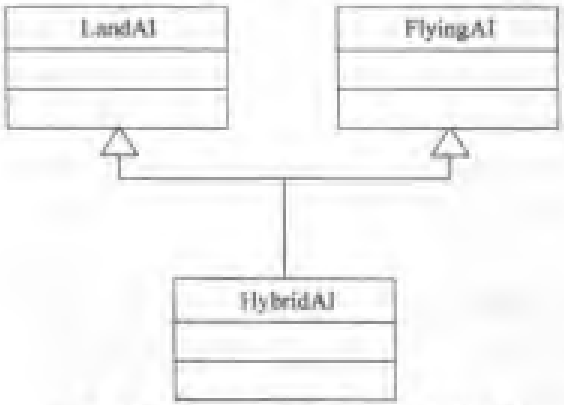


图 12.3 使用多重继承的 AI 类层次

为了增加对多重父类的支持，需要改变几个地方。以前是保留了一个指向父类 RTTI 对象的指针，现在需要为每个父类都保留一个指针。使用 STL 中的 vector 似乎是不错的主意，但是，一来 STL 中的 vector 过于底层，想避免绝对不必要的内存开支，还想管理所有分配的内存。再者，父类的数量在运行时是不改变的，所以在这样的场合下使用 vector 显然是不必要的。

相反，保留一个动态分配的指针数组。在构造函数里，当知道有多少父类的时候，就完成对数组的分配。如果没有父类或者只有一个父类的话，其代码如下：

```
RTTI::RTTI(const string & className):
    m_className(className),
    m_numParents(0),
    m_pBaseRTTI(NULL)
{}

RTTI::RTTI(const string & className, const RTTI & baseRTTI):
    m_className(className),
    m_numParents(1)
{
```

```
    m_pBaseRTTI = new const RTTI*[1];
    m_pBaseRTTI[0] = & baseRTTI;
}
```

这里没有特殊的地方，与以前的相比，这里仅仅是有了更多的信息记录而已。

还需要另外一个构造函数，该函数允许指定想要的父类数量。可以创建一个具有两个父类以及另外一个具有 3 个父类的构造函数。这样做的条件就是构造函数要可以涵盖多重继承中的多数情形，但是，万一有人把多重继承用到极致的话，这些构造函数也可能就不够用了。相应地，可以在构造函数里使用一个可变数字参数，这样就可以避免对父类个数的限制。

这个构造函数需要分配足够的内存，解析参数的变量表，然后把正确的指针复制到数组。

```
RTTI::RTTI(const string & className, int numParents,...): m_className
        (className)
{
    if(numParents < 1)
    {
        m_numParents = 0;
        m_pBaseRTTI = NULL;
    }
    else
    {
        m_numParents = numParents;
        m_pBaseRTTI = new const RTTI*[m_numParents];

        va_list v;
        va_start(v, numParents);
        for( int i = 0;i < m_numParents; ++ i)
        {
            m_pBaseRTTI[i] = va_arg(v,const RTTI *);
        }
        va_end(v);
    }
}
```

在构造函数里使用一个变化的数字参数，一个不太乐观的结果就是不能像另外两个构造函数那样把它巧妙地隐藏到宏里。C（就此而言，C++也一样）不普遍支持带有变量数字参数的宏，这种支持上的不足不允许对每个单独参数进行解析。如果确实想使用宏，那么可以提供少量的宏，以便在有两个或三个父类这样的情形里使用。暂时，还必须明确地

建立构造函数。这也不是太麻烦的事情，只是代码初看起来没有宏那么清晰。

另外在这个例子里还需要注意的，就是需要传递指向父类 RTTI 对象的指针，而不是像前面那样用的是引用。这也要归咎于 C 中对带有可变数字参数的函数的处理方式，C 中不允许这些参数是引用类型的。

```
//In the HybridAI.cpp file
const RTTI HybridAI::rtti("HybridAI", 2, &LandAI::rtti,
                                                &FlyingAI::rtti);
```

只有单一父亲或者根本没有父亲的类可以像以前一样使用宏。在这个例子里，两个 AI 类都没有父亲，因此都可以使用 RTTI_ROOT_IMPL 宏：

```
RTTI_ROOT_IMPL(LandAI);
RTTI_ROOT_IMPL(FlyingAI);
```

剩下惟一需要做的事情就是实现新版本的 DerivesFrom()函数，来检查类的所有父类。

```
bool RTTI::DerivesFrom ( const RTTI & rtti ) const
{
   const RTTI * pCompare  = this;
   if(pCompare == &rtti)
         return true;

   for ( int i = 0; i < m_numParents; ++ i)
         if( m_pBaseRTTI[i]->DerivesFrom(rtti) )
               return true;
   return false;
}
```

这里提供的版本是用递归的方式来遍历所有的父类，这样的不足就是多重函数调用，以及一旦最终找到了一个匹配项时，就需要一层一层地出栈（对应于函数的返回）。不过编写一个更有效的版本来替代带有堆栈的递归版本也是有可能的，而且也避免进行一些额外的函数调用。

即使是使用堆栈的版本在改进以后也会比单继承情况下的相应函数慢。这是因为在遍历所有的父类时，需要执行额外的检查。正是因为这个原因，除了单继承版本以外，在 CD-ROM 中也单独给出了多重继承版本的 RTTI 系统（可以从\Chapter12.RTTIMulti\rtti.dsw 中得到）。

如果知道自己将把设计限制在单继承上，那么应该选择单继承版本。如果决定使用多重继承，那么就可以切换到自定义版本。这些改变是完全透明的，不需要更改任何已有

代码。

12.5　结　论

在这一章里，我们知道，当处理对象的多态性时，有时有必要知道对象的更多信息。这些信息因为是在程序运行时而不是编译时收集到的，所以被称为运行期类型信息。更为重要的是，还了解到，好的可供选择的设计方法不需要对象的任何信息，同时仍可以把它们以多态的形式处理。接着看到了 C++语言提供的运行期信息支持，并学习了它们的用途，了解了它们的局限性。

最后，从头开始，开发了一个自定义 RTTI 系统，这个系统稍加修改就可以在任一已有游戏中使用。还给出了这个系统的几个版本，这几个版本复杂性逐步增加，其中包括一个支持多重继承的版本，多重继承版本是以稍多的内存和性能为代价的。

12.6　阅 读 建 议

以下的这本书是关于标准 C++ RTTI 系统方面的，它包括了所有的细节和一些高级的用法。

Stroustrup, Bjarne, The C++ Programming Language , 3rd ed., Addison-Wesley, 1997.

对于自定义 RTTI 系统，这里没有太多的资料，以下是包括了自定义 RTTI 所有的细节的不多的参考书中的一部分。

Eberly, David H., 3D Game Engine Design, Morgan Kaufmann, 2001.

Wakeling, Scott, "Dynamic Type Information", Game Programming Gems 2, Charles River Media, 2001.

第 13 章　对象的创建与管理

本章内容：

- 对象的创建
- 对象工厂
- 共享对象

大多数时候想象程序运行时，会把程序的运行形象化地理解为程序对象之间的函数调用，数据的传送以及内存的调整。程序的这个方面称为程序的稳态。然而，这只是程序的一个方面。另外一个同样重要的方面，就是程序执行过程中创建对象以及操作对象，这个通常称为程序的动态。毕竟，在开始的时候，C++程序只有静态声明的对象，那么所有其他的东西必须在运行中从头开始来创建。

本章将讨论通过工厂来创建不同类型对象的方法。这些工厂允许用更灵活的方法，而不是普通的 new 调用来创建对象。同时，使用工厂还给以后更容易地进行程序扩展提供了机会。

在这之后，会论述对象创建以后的一些棘手问题。典型地讲就是，代码中会不止一处使用到指向对象的指针或引用，那么如何在不暂停程序的情况下来删除这些对象呢？本章的最后一部分会提供不同的方法来删除对象，这些方法包括通过句柄、引用计数以及智能指针等。

13.1　对象的创建

每个游戏都需要在不同场合里创建对象。只要装载一个新的游戏关卡，就需要创建这个关卡上的所有对象。如果退回到主菜单，就需要销毁这些对象，然后创建与用户接口相关的对象。

但是，对象创建不仅仅局限在加载新的游戏关卡这样的情形中。在游戏进行过程中也可能发生：枪点火就会创建一个子弹对象；子弹撞到墙上就创建一个形成火花和声音效果的对象；人走过水池就产生保证飞溅效果的对象；操纵的个体完成了一个新类型的建筑，当然也必须创建这个对象。还有更多对象创建的微妙形式：按下游戏的某个按钮，就创建了一个向输入处理系统传递的消息；另外的游戏者控制了球，就要创建一个通过网络向你传递的消息包。换句话说，对象创建是无处不在的。

13.1.1　当 new 不满足需要时

在需要创建新对象的时候使用 new 有什么不妥吗？使用 new 来创建对象是最常用的方法。通常该方法除了在编译时需要确定创建对象的类型之外，没有任何不当。如果像下面这样编写代码，游戏就总是创建一个 GameCamera 类型的对象。这里不会发生其他的情形。

```
//This will always be a game camera
GameEntity * pEntity = new GameCamera;
```

在 C++里，可以用多态的方法来操作对象，这就意味着可以不知道对象的特定类型，就可以通过指针，或者其父类类型的引用来得到对象，并对它进行操作（这些在第 1 章里讲述过）。多态的关键就是虚函数。

遗憾的是，C++不允许对象创建时就使用多态。另外，由于 C++非常灵活，所以可以自己编写代码来提供合适的解决方法，但是这需要自己亲自写代码了。

在什么样的情况下，使用 new 就显得不够了呢？最好的例子就是在加载一个新关卡的时候。有一个包含所有信息的文件，这些信息是恢复先前已保存游戏状态所必需的信息。文件包含游戏中所有对象类型信息以及每个对象需要的数据信息（会在第 14 章对象的序列化中更详细的了解）。考虑下面的伪码，就可以看到对动态创建对象需要什么了。

```
//Pseudocode for object loading
for (each object in the file){
   ObjectType type = LoadObjectType();
   GameEntity * pEntity = CreateEntity(type);
   pEntity->Load();
}
```

LoadObjectType 函数返回需要创建的对象类型，然后下面的函数创建正确类型的对象。这确实是想得到的结果。

13.1.2　大 switch 块

不利用虚函数，如何在运行时刻变化代码的行为呢？通常是利用习惯的方式来进行，即使用 if 语句，switch 语句或者是函数指针。下面给出用 switch 语句实现的装载代码：

```
//Pseudocode for object loading
for ( each object in the file){
```

```
    Object Type type = LoadObjectType();
    GameEntity * pEntity;
    switch(type){
          case GAMEENTITY:   pEntity = new GameEntity;   break;
          case GAMECAMERA:   pEntity = new GameCamera;   break;
          case GAMEPLAYER:      pEntity = new GamePlayer;      break;
          case GAMEACTOR:    pEntity = new GameActor;    break;
          case GAMEENEMY:    pEntity = new GameEnemy;    break;
          //add *all * the other possible types of game
          //entities here
          default://Error entity type unknown
    }
    pEntity->Load();
}
```

对于像这样编写出来的代码，如果自己感觉看起来有点不舒服，那么就具有很好的面向对象的意识了。上面的代码在编写中明确依赖于对象类型，这是常见的问题。最直接的后果就是如果增加了新的对象类型，或者是删除、改变了已有对象类型，那么就需要在以后相应地更新程序代码。

另外一个问题就是，使用这样的方式需要程序清楚对象的所有特定类型。本书中反复提到的一个主题就是，一个特定的部分对程序其他部分的信息了解得越少越好，以便更容易进行测试、维护以及在将来进行改进（在了解了程序的物理结构以后，会在第 15 章详细讨论有关大项目的处理问题）。

把需要创建对象的所有信息都集中在一个函数里，其后果就是将来要对程序进行扩展，会变得非常困难。正如在第 10 章和第 11 章里看到的，通常希望的是在游戏发售以后，或者至少是不必重新编译和再分发初始程序的情况下，就可以创建新的对象类型。

例如，假定要给游戏添加一个新关卡，这个关卡里使用了新的实体对象类型，一个可以给游戏增加真实感的动态的天气实体。使用 switch 方法，创建新对象类型的惟一方法就是改变大 switch 语句来检查新类型。如果不能更新可执行程序，那么就糟糕了。如果可以把对象创建的职责分散处理，那么就在改善大 switch 方法的不足的同时，还可以非常容易地扩展程序。

13.2 对 象 工 厂

对象工厂就是基于类型参数创建对象的简单代码。这些代码可以是带有一个枚举类型的简单函数，也可以是具有描述被创建对象的多个参数的完整的类。把所有对象的创建封装到一个工厂具有以下几个优点。

- 如果代码的其他部分需要创建相同类型的对象，那么都可以通过相同的对象工厂

来实现。

- 只需工厂知道所有的对象类型并包含所有的头文件。而程序的其余部分不需要知道有关对象的别的东西。
- 通常，不只想创建对象，经常还会跟踪特定类型的所有对象，或者至少想知道当前分配了多少个这种类型的对象。通过在单一地点创建这些对象，就可以很容易地跟踪这些对象。

在许多不同场合下，必须使用对象工厂的方法创建对象。对象工厂通常也需要保持对象的一些信息，因此它们非常适合单件模式（singleton pattern）。对于每个主要的对象类型，只有一个对象工厂，对象工厂保持了对象所有的相关数据。这样就可以很轻易地从代码的任何地方进行访问了。

13.2.1　一个简单的工厂

从某种程度上讲，在上一节看到的带有大块 switch 声明的函数差不多就是一个非常初步的对象工厂。为了使创建函数与装载函数分离，需要改造这个函数。这样，这个函数就真正变成对象工厂函数了：

```
GameEntity * CreateEntity(ObjectType type)
{
  GameEntity * pEntity;
  switch(type){
        case GAMEENTITY:   pEntity = new GameEntity;   break;
        case GAMECAMERA:  pEntity = new GameCamera;    break;
        case GAMEPLAYER:      pEntity = new GamePlayer;       break;
        case GAMEACTOR:    pEntity = new GameActor;    break;
        case GAMEENEMY:    pEntity = new GameEnemy;    break;
        //add *all * the other possible types of game
        //entities here
        default:pEntity = NULL;
  }
  return pEntity;
}
```

这段代码与在第 10 章看到的创建抽象接口不同实现方面的代码非常相似。

13.2.2　分布式工厂

对于这种情形而言，除了编写了一个非常集中的函数，然后把代码单独分离到这个函

数里，并把这个函数称为工厂之外，是没有任何其他帮助的。仍然有同样难看的代码需要维护，仍然有许多有关所有可创建对象类型的信息需要集中管理。要真正改善这种状况，就需要创建一个分布式工厂。

分布式工厂的隐含思想就是要避免在源代码中硬编码所有可能的对象类型。通过对对象类型的可能类型列表进行动态维护，就可以不费力地在以后增加更多对象类型或者改变已有的对象类型。

到目前为止，一直把某个类的类型与某种类型 ID 联系了起来。只要把 ID 传递到工厂函数，就创建了一个具有正确类型的新对象。这种联系是通过 switch 语句，明确写在代码里的。

现在，计划把这种联系做成动态的。可以建立一个映射，或者是其他类型的数据结构，来把类型 ID 和实际的被初始化的类联系起来。为了实现这一目的，将建立一组非常小的 creator 对象，这些对象的惟一目的就是创建特定类型的对象。对于想创建的每个类类型都需要这些对象的其中一个。以下就是典型的 creator 对象的代码：

```
class CreatorCamera : public Creator
{
public:
   virtual ~CreatorCamera(){}
   virtual GameEntity * Create() const
            {return new GameCamera;}
};
```

为什么要经历这个额外的间接阶段呢？原因就是现在是用 C++的方法来进行处理。在这里，不是把类型 ID 和类类型关联起来，因为这在 C++里是无法实现的。相反，把类型 ID 和特定的 creator 对象联系在一起。这样的话，当需要创建一个新对象的时候，就可以调用 creator 对象的 Create 函数。工厂类像以下这样的代码一样来实现映射：

```
class EntityFactory
{
public:
   GameEntity * CreateEntity(ObjectType type);
private:
   typedef map<ObjectType, Creator *> CreatorMap;
   CreatorMap m_creatorMap;
};

GameEntity * EntityFactory :: CreateEntity(ObjectType type)
{
   CreatorMap :: iterator it = m_creatorMap.find(type);
```

```
    if( it == m_creatorMap.end() )
        return NULL;
    Creator * pCreator = (*it).second;
    return pCreator->Create();
}
```

这里主要利用了 STL 中的 map 容器的优势。只用不多的几行代码，就可以创建和使用复杂且优化的数据结构。如果想多了解 STL 容器方面的内容，可以参考第 8 章。

具有了恰当的工厂实体类后，现在，就可以在不需知道对象特定类类型方面的任何信息前提下来动态创建对象了。

```
ObjectType type = LoadTypeFromDisk();
GameEntity * pEntity = factory.CreateEntity(type);
```

13.2.3　显式 Creator 注册

当然，为了顺利工作，需要告诉工厂支持什么类型的对象并为每个类型传递一个 creator 对象。这些实际上是工厂里的 Register 函数需要做的工作。它允许传递一个对象 ID 和一个 creator 对象，同时，工厂会管理它们彼此之间的联系，并在需要创建那种类型的对象时去使用这些联系。

```
bool EntityFactory :: Register(ObjectType type,Creator * pCreator)
{
    CreatorMap :: iterator it = m_createMap.find(type);
    if(it != m_creatorMap.end()){
        delete pCreator;
        return false;
    }
    m_creatorMap[type] = pCreator;
    return true;
}
```

现在注册一个新类型的对象就很简单了，只需要创建一个新的 Creator 类然后调用 Register 函数即可。

应该在什么时候调用 Register 函数呢？这依赖于每个项目的具体情况，但是通常是在系统初始化阶段需要创建那些对象时调用 Register 函数。在游戏实体的例子中，比较好的机会应该是在游戏刚被初始化，还没有创建或加载任何实体之前调用注册函数。这样，游戏的初始化函数看起来应该如以下代码所示：

```
factory.Register(GAMEENTITY, new CreatorEntity);
factory.Register(GAMECAMERA, new CreatorCamera);
factory.Register(GAMEPLAYER, new CreatorPlayer);
factory.Register(GAMEACTOR, new CreatorActor);
//_Register the rest of the entity classes
```

已经把可以创建的对象类型信息分散开了。一个插件可以在初始化以后通过调用Register函数来增加新的对象类型。当那种新类型的对象被代码的任意其他部分装载或者创建的时候，就会调用正确的工厂函数。

13.2.4 隐含 Creator 注册

为了使用的便捷性，可以把系统做得更深入一步。如果处理得当，就可以避免显式地调用 Register 函数。这样就会使得整个系统更加透明，增加新类型就只是创建新 Maker 对象的事情了。

在编写代码时，可以让 Maker 对象的构造函数完成在工厂里的注册。这样的话，所需要注意的就是要确保至少初始化一个 creator 对象，以便调用它的构造函数。通过静态地声明一个对象，然后让 C++初始化代码为我们处理构造函数的调用。现在，Creator 对象的代码如下所示：

```
class CreatorCamera : public Creator
{
public:
   CreatorCamera() { GameFactory::Register(GAMECAMERA,this);}
   virtual ~CreatorCamera() {}
virtual GameEntity * Create() const
                          {return new GameCamera;}
}instance;
```

注意在代码中，构造函数自己完成了注册工作，特别地，对象 instance 是在类声明之后直接生成的。有了这样的安排，就不再需要调用 Register 函数了。正如将会看到的那样，不调用 Register 函数得到的方便可能没有因此引起的问题多。

需要牢记的第一点，就是 creator 类在静态初始化时刻形成。遗憾的是，如果静态变量处在不同的文件中，那么 C++不能确保这些变量的初始化次序。因此，必须确保所有静态变量的初始化不能依赖其他已初始化的对象。如果需要其他对象，那么可能应该在需要的时候使用显式初始化声明。

第二个问题就是对于 creator 注册方面没有太多控制。只要它们被编译器处理了，不论

是否想让它们注册，它们都会被注册。这对于游戏实体而言可能是件好事。但是如果是在注册用于图形资源的 creator 的话就不太可以接受。可能计划让不同的平台有不同的 creator，或者是在相同平台的不同图形 API 函数有不同的 creator。如果是这样的话，就想在运行时刻基于用户的硬件或者一些配置文件，可以选择是否注册 OpenGL creator 或者 Direct3D creator。

即使前面提到的任何一个问题对你而言都不是问题，那么，对于显式 creator 对象的注册，这里还有一个讨厌的地方。为了保证可执行程序尽可能的小，有些编译器会把没有在任何其他地方调用的代码认为是无用代码，从而在链接时会清除这些无用代码。这样做自然有其有利的一面，但是当编译器过度清除无用代码时，就会有一些不好的结果。例如，看一下 CreatorCamera 类。在程序中既没有任何地方直接使用到那个类，也没有使用到创建的静态变量 instance。还有更让我们烦恼的就是，一些编译器把 instance 变量一起清除掉，这样的话就根本不会进行 creator 类的注册了。

可以快速地知道自己的编译器是不是属于这种情况。有些时候，只是在 creator 类位于静态库里时，这个问题才会出现，这通常也是大项目下的组织情形。或许能够找到解决方法来强迫编译器包含这些代码，但是这种处理过程不会是愉快的事情，同时也会因为平台的差别而需要不同的处理。强迫编译器来使用同一文件中的其他变量或许就足够了，另外有些编译器还提供了编译器特定的#pragma 指示来强行连接文件中的所有符号。也可以关闭无用代码清除选项，但是这可能导致可执行程序的尺寸增加得太大。总而言之，除非确实认为上面提到的所有不足都不会影响到自己，否则最好的方法可能就是简单地显式声明，然后手工调用 Register。

12.2.5　对象类型标识符

讲述到现在，一直彻底掩盖了如何指定对象类型这样的细节。一直是在使用类型变量 ObjectType，但是这些变量实际是什么呢？通常，这要取决于特定需要。

指定对象类型最简单的方法就是使用包含类名称的字符串。因为字符串的内容可以从调试器里读取出来，所以字符串是非常容易测试的，同时也容易实现每个类具有一个独立的字符串标记。甚至可以把 RTTI 系统（既可以是标准 C++也可以是自定义的）返回的标识类名称的字符串拿来使用。另一方面，使用字符串也有不足的地方，首先是保存字符串需要更多的内存（既然对每个类类型只需要一个字符串，那么代价不是很昂贵）。但是更重要的是，与整数或者枚举类型相比，字符串的比较操作显得要慢一些。

如果对象大多是在游戏关卡装载时创建出来的，那么这就不会是一个问题。任何性能方面的损耗都会完全被 IO 访问硬盘或者 CD-ROM 所掩盖。另外一方面，如果想在运行期使用工厂来创建很多对象的话，那么那些额外的性能损失在累加之后的影响可能就很显著了。

可选择的方法就是使用整数或者枚举类型。使用枚举看起来是一个不错的主意：它不

仅小，也可以快速实现比较。与字符串一样，同样可以很容易地从调试器里读取出来。这些都是事实，但是一旦定义了枚举，那么就不可能扩展它们了。记住，想利用插件增加新的对象类型，因此要是使用枚举来描述所有的对象类型，那么就返回到初始的问题上了，即把所有可能的类型固化在了一个地方。

因此剩下的选择就是使用整数或者其他轻量级对象来惟一地描述对象类型了。为了其保证惟一性，尤其是不从插件的观点考虑的话，因为不知道这里有多少其他类型的对象，所以不能只是对每个对象类型的标识符施加增量（比如加 1）。最好的方法就是尽量在类的基础上生成一个伪惟一的 ID，例如使用类名称的 CRC 或者类静态变量的内存地址。如果需要一个更牢靠的方法，那就是生成完整的 UID（UID 是这样一些数字的组合，组合后的数字有惟一性）。

12.2.6 使用模板

如果想通过一个工厂生成几个完全不同类型的对象，例如，像游戏实体，还有资源、动画或者其他没有与游戏实体共享一个基类的任何东西，那么应该怎么办呢？只为每个组创建一个工厂。游戏实体有自己的工厂，资源也有自己的工厂。需要在正确的工厂里注册 creator 对象并调用正确工厂里的 Create 函数。

假定在编写一个工厂以后，意识到第二个工厂的代码与第一个工厂不是截然不同的。事实上，可能确实是相同的代码，但是使用的却是不同的基类。对于所有的 Maker 对象也是相同的情形。读到的情形看起来是不是很熟悉？这是使用模板的好场合。需要相同的代码，但是需要变换代码作用的类。下面给出模板化工厂的代码：

```
template<class Base>
class Factory
{
public:
   Base * Create ( ObjectType type);
   bool RegisterObjectType type, CreatorBase<Base> * pCreator);
private:
   typedef std::map<objectType, CreatorBase<Base> * > CreatorMap;
   CreatorMap m_creatorMap;
};
```

creator 类也必须被模板化。先前，所有的 creator 都是从共有的 Creator 类继承过来的，这也是在工厂里存放的指针类型。既然 creator 基类的 Create 函数返回一个指向正在创建的对象的基类的指针，也需要模板化 creator 基类自己，以便返回的指针类型可以以正确的方式相匹配。

```
template<class Base>
class CreateBase
{
public:
   virtual ~CreatorBase() {}
   virtual Base * Create const = 0;
};

template<class Product, class Base>
class Creator : public CreatorBase<Base>
{
public:
   virtual Base * Create() const { return new Product;}
};
```

工厂函数的实现正像前面给出的非模板化版本，只是相应带有通用类型参数而已。参考 CD-ROM 上的源代码来详细了解模板实现。Visual Studio 工程位于\Chapter13.Object Factory\ObjectFactory.dsw。

有了这些模板，形成新的工厂和 creator 就容易了：

```
Factory<GameEntity> factory;
Factory.Register(GAMECAMERA, new Creator<GameCamera, GameEntity>);
Factory.Register(GAMEENTITY, new Creator<GameEntity,GameEntity>);
```

另外使用起来也和以前一样简单：

```
GameEntity * pEntity = factory.Create(GAMECAMERA);
```

只要所有的对象类型用一个默认的构造器创建，那么就不需要在工厂里做额外的处理工作，使用模板化工厂和 creator 大概就是一个很好的方法了。然而，如果对象需要特殊的创建方式，这种方式就显得不够了。或许需要把一个参数传递到它们的构造函数，或者是对象创建以后需要立即调用一个函数。这些都是模板化方法不可以实现的。类似地，模板化方法不允许你在自己的工厂里保留对象列表，或者关于对象的一些统计信息。在这样的情况下，非模板化解决方法可以提供需要的更多灵活性。

13.3　共 享 对 象

创建对象只是乐趣的一半。一旦对象已经创建出来，就需要知道它们的存在，并管理它们的生命周期，然后正确地删除它们。一旦代码里有不止一处引用到同一对象，那么这些操作就会显得更困难。

C++没有任何自动垃圾回收机制。因此，程序员需要手工管理内存。有了自动垃圾回收的支持，程序员就不需要担心分配了什么内存，释放了什么内存。运行期的代码负责释放程序不再使用的所有内存。当然了，垃圾回收有自身的问题，例如在不合适的时间会占用程序的处理时间并使得失去了对渲染帧的垂直同步。

没有垃圾回收机制，就意味着必须小心翼翼地对待已分配的内存和创建的对象。需要跟踪它们，并在不再需要的时候删除它们。遗憾的是，事情比这还复杂。首先来看看以下的代码：

```
//Create a new explosion and point the camera to it
GameEntity * pExplosion = new GameExplosion;
Camera.SetTarget(pExplosion);
//…
//Some time later, the explosion dies
delete pExplosion;
```

camera 类保留了一个它聚焦的实体的指针。那么那个实体删除以后会怎么样呢？镜头类怎样知道这些呢？指针只是一个内存地址，这个地址没有改变过，所以指针看起来仍然是相同的。但是现在那个内存不是不属于这个进程，就是被完全不同于初始实体的东西替代了。因为指针指向的内存位置已经不存在或者已经完全改变了，所以称为悬指针。

这就是共享对象的基本问题。本章剩下的部分将提出几个方法来处理程序中的共享对象。

13.3.1 无对象共享

通常，对于一个问题，可能的解决办法之一就是首先要避免出现这样的问题。如果绝对避免共享对象的话，那么问题也就不存在了。遗憾的是，彻底不使用共享对象是不可能的，或者是不大满意的，有时使用共享对象还是有意义的。

在大多数游戏里，如果两个游戏对象占用同一个昂贵的资源，那么让两个对象来共享资源，而不是在内存中保留两份重复的备份就显得有意义了。例如，给定类型的所有建筑物可以共享材质，所有坦克可以共享相同的发动机轰鸣声。在游戏里给每个实体都创建同一资源的新备份显然是一种浪费。

在前面的例子里，camera 需要指向一个已经存在的游戏对象。对于 camera 对象而言，自己具有对象的备份是完全没有意义的。因为其他对象可能离开，那么镜头就只是指向了对象的固定备份。这种场合下仿佛是必须使用对象共享了。

然而，如果对象共享不是必需的，那么就要避免使用它。这样会使代码更清晰、更有效，也更易于维护。没有必要在普通指针可以实现功能的时候引入复杂性。

例如，一个 camera 实体可以保留一个指向旋转矩阵的指针来作为一个私有的成员变量。

指针不需要暴露给游戏里的其他部分。只有 camera 知道它的存在，并具有对它的完全控制。当 camera 决定删除矩阵的时候，这里不会有其他指针指向它，因此就不会碰到任何问题。camera 自己可能在镜头已经被删除的时候就把指针置为 NULL，因而 camera 就知道不再去访问那个指针了。

13.3.2　忽略问题

有些时候，对于简单问题及小项目而言，鸵鸟方法可以有效地工作，即故意不看将来的危险从而忽略问题。诚然，这样的方法会引起潜在的麻烦。但是既然项目很小，所以就可能会预见到没人会在共享指针删除以后再使用它了。这有点像在玩火，但是如果不被烧着，那么就是毫无危害的。

对于任何大小合理的项目，可能是技术演示或者是快速建立的宣传原型，这种方法显然不是值得推荐的。另外，如果团队包括多位程序员，其他程序员如果忘记删除对象的隐含管理规则，则会意外地导致 BUG，所以这个方法就一定会失败。

事实上，对于在运行时刻不使用动态分配的程序，可能也能够侥幸使用鸵鸟方法。如果一切都是静态分配的话，对象就决不会删除，那么指针就会保持为有效的。通常在使用对象前，需要核对通过指针访问的对象是活动的，或者是有效的。那么，如果游戏是这样组织的，那么这个方法就非常适合了（参见第 7 章中关于静态和动态内存分配策略的比较）。

因鸵鸟方法的使用而出现问题也是不容易解决的。这些 BUG 不是简单跟踪的问题。在调试器里去找出谁在过去某个时刻，删除了一些内存可不是什么愉快的工作。另外，确定冲突是由删除的共享指针而引起的也不是一件容易的事情。

当项目比预期的要大时，尽力把项目从忽略共享对象问题转化为采用本章讨论的其他实现方法之一的话，这种转换就会显得非常棘手了。这种转换是有可能的，但很耗费时间。尤其对里程碑方法和时间变得很宝贵的时候，预先计划与规划好完全可以节省大量时间。

13.3.3　由共享对象的拥有者负责共享

处理共享对象问题时，一个可能的方法就是指明一段代码，或者在例子里是一个特定的对象，来担当共享对象的所有者。可以有许多来自不同对象的指针和引用使用一个共享对象，但是这些对象中只有一个是共享对象的拥有者。

拥有者只负责创建、管理以及最终销毁共享对象。就像前面的方法一样，实在没有什么办法可以不让对象的非拥有者来删除共享对象。这更多的是一个保证一切顺利进行的代码规则。因为这个方法比前面的方法更具体，所以只要每个人小心点，就可以毫无困难地把这个方法用于大型团队项目。

这样看来，拥有者现在就可以自由地删除对象了，但是对象的非拥有者对象指向对象的指针会怎么样呢？这些指针必须知道对象在什么时候被删除了，以便将它们的指针更新

为 NULL 或者只是改变指针的状态，这样就可以保证那个指针不再被使用了。

利用类似于 Observer 模式这样的安排，就可以实现提示，对象的非拥有者是订阅主题提示的观察者，这些主题可以是拥有者或共享对象本身。当共享对象被删除的时候，所有的观察者收到带有相应的消息。

正是因为共享对象有所有者即可，所以不需要所有者始终保持不变。只要涉及的所有对象同意，就可以根据需要把一个对象的拥有权转移。例如，在多游戏者战略游戏里，一个游戏者控制的所有单元可以由这个游戏者拥有。如果游戏者碰巧断开了，那么可以创建一个 AI 游戏者并把所有刚才那个游戏者的单元拥有者转移给 AI 游戏者，这样每个人就可以继续打游戏了。

这个方法需要决定谁是拥有者，以及如何来组织共享对象的生命期。该方法还要求编写对象的非拥有者时，可以处理随时被销毁的共享对象。

最后，如果有许多观察者，而且对象频繁创建与删除的话，因通知每个观察者消耗的性能开销，以及保存所有观察者所需要的内存都会很显著。对这个方法，效果最好的对象就是相对大型、而又不在每帧里多次销毁的对象。

13.3.4 引用计数

另外，引用计数使用的也很广泛。在这种方法里，不需要共享对象的所有者了。替代的是，只要有人需要这个对象，那么它就会保留下来。只要对共享对象的最后一个引用不存在了，就知道没有人再使用这个对象了，然后删除对象。引用计数就像一个基本的垃圾回收系统。

1. 实现

要能够使用引用计数，共享对象就需要实现两个函数：AddRef 和 Release。当代码的任一部分获得对象的指针时，AddRef 被调用。当代码的任一部分结束对对象的使用时，不是删除它，而是调用 Release。

被引用计数的对象知道自己有多少引用。每次调用 AddRef，就会增加引用计数。每次调用 Release，引用计数就减少 1。如果引用计数到达零，那么对象就被删除。

这实际就是 COM 管理共享对象的方式。如果使用 DirectX API，那么也会经常使用这些调用，因为 DirectX API 是基于 COM 系统的。

为了更易于使用，可以把所有的引用计数功能放到一个类里，并且允许继承该类的任意类可以自动地成为可引用计数的。

```
Class RefCounted
{
public:
  RefCounted();
```

```
    virtual ~RefCounted() {};

    int AddRef() { return ++m_refCount; }
    int Release();
    int GetRefCount() const {return m_refCount;}

protected:
    int m_refCount;
};

int RefCounted::Release()
{
    m_refCount--;
    int tmpRefCount = m_refCount;
    if(m_refCount <= 0 )
        delete this;
    return tmpRefCount;
}
```

要使用它，只需要一个继承 RefCounted 的类。如果想把这个类做得更安全一些，可以把它的析构函数声明为受保护类型。这样就可以确保对象销毁的惟一方法就是通过 Release 调用。利用内部代码来直接销毁对象的任何企图都会导致编译期发生错误。

```
class GameEntity : public RefCounted
{
public:
    //...
protected:
    virtual ~GameEntity();
};
```

在 RefCounted 实现中值得关注的有两个有趣的地方。首先，注意 AddRef 和 Release 函数返回的是整数。这些返回值就是在调用发生以后的当前引用数量。有时为了知道一个对象是否被销毁，对于处理被引用计数的对象的代码是需要的。

第二个有趣的地方就是 Release 的调用。大概以前没有看到过 delete this 这样的代码。看起来从成员函数内部调用这个有点危险，但是它是非常安全的。对象删除的是它自己，这与别人删除它是毫无区别的。必须注意的是在 delete 调用以后不要再调用其他函数或者访问任一成员变量了。这也是要迂回地在堆栈里保留一个临时变量以记录引用计数的值的原因。如果想直接返回 m_refCount 的值，那么一切在引用计数到达零以前都会很好工作，在计数为零的时候对象将被删除。然后会试图读取对象成员变量之一，正如可以想到的一

样，程序将会崩溃。但是如果返回的值和内存地址都保存在堆栈里，那么就能够安全地从函数返回了。

2. 建议

引用计数有什么缺点吗？遗憾的是，几乎没有。可能看起来这个方法是彻底安全的。只要引用计数一到达零，对象就会被删除。不需要任何人记着在某处删除对象并通知其他对象。一旦 AddRef 和 Release 调用出现不平衡，因为它们的引用计数仍然保持上升，所以对象也决不会消失，或者是因为它们的引用计数偶然到达零而被永久删除掉。捕获不匹配的 AddRef 和 Release 调用也不是容易的事情，而通常只要一个不匹配的调用就可以导致系统崩溃。

尽管如此，引用计数与试图忽略问题相比还是一个非常稳健的系统，它具有的优势是无须指定所有者来管理对象。对于共享对象不具有严格所有权的场合，这个方法就显得非常合适。一个理想的例子就是游戏中使用的图形资源、纹理以及动画等，它们可以保留一个引用计数来记录有多少游戏对象使用自己。只要使用指定纹理的所有游戏对象被删除了，材质也就被销毁了。

这就带来了引用计数的第二个不足，对象可能有点过于容易被销毁。可能刚释放一种纹理，但是就将生成一个需要那种纹理的新实体。这时宁可让纹理对象停留在内存中，但是现在，被迫要再从磁盘装载材质。可以通过一个资源管理器来多保留一个对所有纹理的引用，利用这样的方法可以解决刚才那个问题。这样的话，即使游戏中没有任何实体使用某特定纹理了，资源管理器也可以根据需要保留它。

该方法的最后一个缺点就是可以导致冗长、笨拙的代码，这是一个比较重要的问题。这类似于异常处理中通过函数返回值来检查错误的类型。可能检查错误的代码比做实际工作的代码行还要多。同样的情况可以发生在引用计数上，代码会很快到处充满了 AddRef 和 Release 调用。

使情况更为糟糕的是，带有应用计数对象指针的函数通常把那些指针作为参数而不是直接返回指针，这样也使得代码更费解也更冗长。

看看下面的代码：

```
GameEntity * GameCamera :: GetTarget()
{
   //We assume that the target always exists
   m_pTarget->AddRef();
   return m_pTarget;
}
```

现在，编写如下的代码可能对一些人来讲是非常有诱惑力的：

```
cout<<pCamera->GetTarget()->GetName();
```

GetTarget 函数的返回值直接从临时变量里使用。并且在声明之后，就不再访问它了。这意味着对目标对象的引用计数增加了，但是从不调用 Release。为了避免这样的问题，比较典型的函数形式如下：

```
void GameCamera :: GetTarget ( GameEntity *& pEntity)
{
   //We assume that the target always exists
   m_pTarget->AddRef();
   pEntity = m_pTarget;
}
```

同时按照下面的方式来使用：

```
GameEntity * pEntity;
pCamera->GetTarget(pEntity);
cout << pEntity->GetName();
pEntity->Release();
```

以前的一行现在编写成了四行代码。还有，有时为了得到更稳定的方法来管理共享对象，这样的方式是值得采用的。

13.3.5　句柄

使用共享对象问题的根本原因就是会有多个指针指向同一对象。句柄能够阻止这种情形的发生，因此，就不会再有对象删除和遗留悬指针的任何问题。

句柄的原理就是使用某类标识（一个句柄）而不是共享对象的指针。那么，任何时候想直接对共享对象操作的话，就要求共享对象的所有者提供一个具有相应确定句柄的对象指针。通过指针对这个对象执行操作，但是当操作结束以后，会丢弃这个指针，不会保留它。

这样，每个共享对象确实只有一个指针，即指向对象所有者的指针。无论何时有人要使用对象的话，它们需要首先传进句柄。如果对象已经被删除了，可以返回一个 NULL 指针。这样他们就知道对象不存在，从而可以进行相应处理。

句柄可以只是一个普通的整数。因为整数小，所以可以有效实现复制与传递，使用起来会显得非常方便，但是句柄也可以是包含较大范围的惟一值。要让这种方法起作用，就必须把每个句柄惟一地映射到每个对象。

如何保证给予实体的句柄是惟一的呢？正确地使用一个 32 位数字然后每次需要一个新实体的时候在这个数字上增加，就可以提供 43 亿个不同的数字。这对于任何游戏都远远

够用了。以下是在真实情形下使用句柄处理纹理的代码：

```
typedef unsigned int Handle;
Handle hTexture = CreateTexture("myTexture.tif");
//…
Texture * pTexture = GetTexture(hTexture);
if(pTexture != NULL){
   // The texture was still around ,do something with it
   //…
}
```

处理32位数字句柄的一个副作用就是不可能从任意随机数中验证有效的句柄。如果内存偶然被破坏，复制句柄时出现问题，或者（非常可能的情形）一个句柄变量从来没有初始化过，那么就在其中包含随机数据。在这样的情况下，句柄就会成为一个问题。如果有方法可以鉴别有效句柄和无效句柄不是更好吗？

可以牺牲头部的几位字节，用它们来保存“惟一的”位模式。可以使用前面的8位，这样仍然留下2^{24}个不同的句柄。甚至可以把16位作为惟一标识符，这样仍然还有60000个句柄可以标记实体。特别地，如果只想对每个主要的共享对象类型使用惟一句柄，这些数量对多数游戏是足够多的了。例如，所有的资源可以使用一个句柄系统，但是所有的游戏实体可以使用另外一个。实际上并没有真正的惟一位模式，但是一个随机数能够匹配位模式的几率是非常小的，并且使用的位数越多匹配几率就越渺茫。

当想在句柄和指针之间转换的时候，程序可以首先验证句柄有正确的位模式，然后如果无效的话就可立刻断言（参见第16章，防止程序崩溃的使用断言这一部分）。如果性能是一个问题的话，验证可以只在调试模式下使用，而在发行模式下避开它。

因为在得到指向共享对象指针的时候需要转换步骤，所以句柄可能稍微有点麻烦。然而，对于不需要经常接触到潜在指针的任务而言，句柄是非常有效的。例如，在代码里不应该有许多地方试图得到网格纹理的指针。那可能只是在装载网格以及渲染的时候发生的事情，因此在这样的场合下就可以使用句柄。

当共享对象被删除，而以后需要重新创建的时候，句柄也是非常有效的。如果确保新创建的对象映射到对象以前具有的同一句柄上，那么所有以前的句柄又成为有效句柄。在游戏里，当游戏者移动到不同的位置，就需要来回缓存资源，这种情况下句柄也是非常适合的。

句柄到指针的转换应该以更快的方式实现。通常，好的实现就是哈希表或者map，但是也可以考虑使用指针数组（或者vector），以使得每个句柄正好是数组的索引。主要的性能开销来自于多了一个额外的间接层，需要多做一层转换。

13.3.6　智能指针

句柄、引用计数以及所有讨论过的处理动态对象的不同方法，都是试图解决一个问题：如果共享对象被删除，则有悬指针存在。如果指针知道它正指向一个被删除的对象，或者共享对象只要有指针指向它就不会被删除，这样的方式不是更好吗？这正是智能指针提供的功能。

最好称 C++指针为“哑”指针。它们实际上做的工作很少。它们指向一个内存位置，而且知道想从那里发现什么类型的数据。并不知道实际上在那个位置进行着什么，同时它们无法意识到人们正在使用或者复制它们。

可以利用 C++的灵活性和强大来生成对象，让这些对象感觉起来就像指针，但是这就要求做一些额外的工作。

智能指针确实没有一个确切的概念。不同智能指针可以完成不同的工作：检查内存的有效性，保留引用计数，在复制指针时使用不同类型的复制措施，保留统计信息，或者在指针自己被删除的时候删除指向的对象。

在第 5 章讲述异常处理时，实际上已经见过一些智能指针了。那里有一类智能指针就是 auto_ptr，这个指针可以在自己被删除的时候也同时删除指向的对象。要正确地释放所有资源，甚至在异常和栈回卷的场合，这个能力也特别有用。

智能指针要可以有效工作就必须像真正指针那样运行。也就是说，可以在 C++里用以前使用指针一样的方式来使用智能指针。应该能够使用->和*有效地复制，并且还是类型安全的，而且应该很小且不占用很多内存。能够实现上面的所有目标。

在共享对象中可以用到的智能指针主要有两个不同类型。两个类型都实现了前面已经讨论的方法，但是它们用类似对象的指针封装起来，这样就更为安全了，同时确实有助于减少许多在直接处理句柄和引用计数时的无关代码。

1. 基于句柄的智能指针

基于句柄的智能指针就是句柄的一个封装。它不需要具有任何其他数据。其方便性是因为在多数平台上，可以使得基于句柄的智能指针就像正常指针那么小。

指针将处理所有的跑腿工作。大家可能记得，为了使用句柄是如何必须首先把句柄转换为实际指针的。智能指针将完成这样的工作。只是把它作为一个正常指针使用就可以了。要这样做，需要重载 C++中一些深奥的运算符，例如->和*。这里是一个基于句柄的智能指针的骨架：

```
class EntityPtr
{
public:
  EntityPtr(Handle h) : m_hEntity(h) {}
```

```
    bool operator == ( int n) {
                        return (n == (int)GetPtr(m_hEntity);)
    bool operator !=(int n) { !operator == (n);}
    GameEntity * operator->() { return GetPtr(m_hEntity);}
    GameEntity & operator *() { *return GetPtr(m_hEntity);}

Private:
    Handle m_hEntity;
};
```

由于重载了运算符，那么现在留意如何把智能指针作为正常指针对待。

```
EntityPtrptr( GetEntity() );
if(ptr != NULL) {
    cout << ptr->GetName();
    const Point3d & pos = ptr->GetPosition();
}
```

因为已经重载了运算符==，该运算符可以检查一个相应的指针是否存在而且非空，所以甚至能够像通常处理指针那样检查指针是否为 NULL。

上面的智能指针只对 GameEntity 类型指针起作用，但是确实想对其他指针使用它，包括资源指针或者使用句柄的任何指针。既然惟一需要改变的就是指针类型，那么可以使用模板来创建想要的指针类型：

```
template <class DataType>
class HandlePtr
{
public:
    HandlePtr ( Handle h):m_hEntity(h){}
    bool operator == (int n) {
          return (n == (int) GetPrt ( m_hEntity);}
    bool operator != (int n) { !operator == (n);}
    DataType * operator->() { return GetPtr(m_hEntity);}
    DataType & operator*() {* return  GetPtr(m_hEntity);}

private:
    Handle m_hEntity;
};
```

要使用它，只需要用想指向的类型来实例化模板即可：

```
typedef HandlePtr <Texture*> TexturePtr;
TexturePtr  pTexture = CreateTexture('smiley.tif');
```

很快就具有了指针的便利性和句柄的安全性。这些就是从智能指针得到的好处。

2. 引用计数智能指针

引用计数智能指针把引用计数方法封装到一个智能指针里。从概念上讲，每次给对象创建一个智能指针，引用计数就增加。每当对象销毁了，应用数量就减少。在某种程度上，在引用计数智能指针自身生命周期基础上，是通过调用 AddRef 和 Release 自动实现的。通常，如果共享对象的引用计数到达零，则删除指针。

实际上，事情比这有点复杂。必须了解复制智能指针的机制并且正确处理，甚至想把引用计数移出共享对象之外，放置到智能指针自身内部（为了更详细地了解不同的实现，可以参看本章末尾给出的阅读建议）。

Boost 智能指针库提供了不同意图下可以使用的不同类型智能指针。其中之一就是 shared_ptr，这个指针确实实现了应用计数方法。正如对待基于句柄的智能指针那样，它们也被模板化，可以安全地对任意指针类型使用。

3. 建议

智能指针是两个方法中最好的。如果准备花点时间，并从开始就计划着如何把智能指针融入引擎的话，它们确实是一个理想的解决方法。

从头开始实现一个智能指针是非常简单和明了的。但是如果实现的机制复杂，则使它在所有情况下都正常工作就会比较困难（例如，临时指针，把一个指针从栈里复制到堆里）。

如果需要的功能已经由 Boost 库的某个智能指针提供了，那么直接使用就是了。但要确保自己认真阅读文档，从中了解指针实际的内存和性能成本是什么，在被复制、删除或者进行传递时的行为是什么。

在有些情形下，智能指针的表现不总是和正常哑指针一样的，情形之一就是进行不同类型强制转换的时候。在 C++中，从指针到子类的变换到指针到父类的变换都是自动发生的。除非提供了特殊的构造函数，否则同样的转换不会发生在智能指针上。最通用的方法就是通过提供特殊的模板化函数来执行强制转换。

13.4　结　　论

在这一章里，已经看到如何使用对象工厂灵活地创建任一对象类型。对象工厂允许创建一个对象，这些对象是在运行期决定而不是在编译期决定的。这就使创建和加载对象变得非常容易，还可以动态地扩展游戏中的对象类型。

接下来看到在 C++里，使用共享对象是如何产生问题的，如果多个对象都指向一个共

享对象，那么哪个对象负责删除它？而且更重要的是，当那个对象删除以后会怎么样呢？

给出了几种不同的技术来处理共享对象，这些技术包括：

- 不共享任何对象
- 不关心对象共享和对象销毁
- 由所有者处理对象的销毁并负责通知所有指向共享对象的对象
- 利用 AddRef 和 Release 函数来应用引用计数
- 为避免保留多余的指针，使用句柄
- 使用智能指针来实现引用计数或者句柄的功用，仍用指针语法

13.5 阅读建议

在对象工厂方面有一些不错的参考资料。特别地，Alexandrescu 编写的参考书非常深入地讲解了模板化工厂的实现细节。Design Patterns 这本书也包括了对 Observer 模式的通用描述。

Gamma, Erich, et al., Design Patterns, Addison-Wesley, 1995.

Larameé,FranCois Dominic, " A Game Entity Factory", Game Programming Gems 2, Charles River Media, 2001.

Alexandrescu, Andrei, Modern C++Design, Addison-Wesley,2001.

接下来的参考资料里较好地讲述了如何用句柄进行资源管理：

Bilas, Scott, " A Generic Handle-Based Resource Manager," Game Programming Gems, Charles River Media, 2000.

接下来的这些书更加详尽地对引用计数进行了解释，并且深入钻研了实现健壮智能指针的复杂性。

Rogerson, Dale, Inside COM, Microsoft Press, 1997.

Eberly,David H., 3D Game Engine Design, Morgan Kaufmann, 2001.

Meyers, Scott, More Effective C++, Addison-Wesley, 1996.

Alexandrescu, Andrei,Modern C++ Design, Addison-Wesley, 2001.

Hawkins, Brian, "Handle-Based Smart Pointers," Game Programming Gems 3, Charles River Media, 2002.

最后，Boost 库提供了几个智能指针类以及每个类的详细文档。

C++ Boost Smart Pointer Library, http://www.boost.org/libs/smart_ptr/index.htm.

第 14 章　对象的序列化

本章将讲述：

- 游戏实体序列化概述
- 游戏实体序列化的实现
- 组装起来

序列化是指将一个对象存入介质当中以便日后进行恢复，这些介质可以是任何形式的数据存储设备，包括内存、硬盘或网络管道。

序列化是游戏的重要环节，至少在游戏开始的时候，需要加载游戏的关卡、装备资源和游戏状态。大多数游戏都允许在游戏进行中进行存盘，这样玩家日后就可以继续从当前状态开始玩起。

与其他语言不同，C++并没有提供内置的序列化工具，因此，需要手工实现所需的功能。本章主要来看一看如何在游戏中提高序列化的效率，检查序列化带来的问题，以及处理游戏实体和游戏资源的一些建议性的意见。

14.1　游戏实体序列化概述

几乎所有的游戏都需要存储游戏实体的状态。这在很多时候表现为游戏中间存储点或检查点的形式，这些“点”允许用户在将来的某一时刻返回并从这点继续开始。即使有些游戏中不提供在游戏中间保存的特性，它仍然需要在每一关开始的时候为所有游戏实体建立初始状态。大多数情况下，这些初始状态是通过关卡编辑器生成的，设计者可以通过这些工具在游戏中生成游戏实体，设置参数以及它们之间的关系，并对它们进行保存从而生成新的游戏关卡。

14.1.1　实体与资源

游戏实体是代表游戏中某些特定事物信息的对象，它们是与我们进行交互的对象：向我们射击的敌人，游戏中所要实现的目标，观察环境的镜头，所有的这些都应该随着游戏的进行而发生变化，要注意对它们进行存储以便保存整个游戏的状态。

区分游戏实体和游戏资源是很重要的。游戏资源是游戏实体以外的部分，游戏资源在游戏过程中不会发生改变，它不包含任何与某一实体相关的特定信息，它们的存在只是为

了供实体使用，例如纹理、模型、声音和脚本文件等。

区分游戏实体和资源的关键就在于，资源一旦载入一般就不会发生变化（它们在某些特定的环境下也有可能发生变化，在后面会讨论这种情况）。不需要保存资源来获得游戏的状态。关卡开始时，这些资源从光盘中载入，并不再改变。这就意味着，只要重新载入游戏实体就可以恢复以前的状态，因为所需的资源都已经在内存中了。

实体和资源代表了完全不同的两个方面，实体既需要载入也要被存储，而资源则只需要被载入。实体往往是相对较小的对象，它们包含了不同的数据信息，并且它们之间都是紧密联系的。相反，资源通常规模较大，类型各不相同，相互之间也没什么内部联系，鉴于实体与资源的不同性质，本章将只讨论游戏实体的序列化问题。

14.1.2 简单却不实用的解决方案

开发游戏过程中通常会遇到很多难题，游戏实体序列化是其中的一个。它在开始的时候看起来很简单、很直接，但用起来却很复杂、很费神，很难保证它能够正常工作。这个问题初看起来似乎不值得一提，就是遍历游戏中的每一个实体，把对象信息存储到磁盘文件中，然后再读取文件以恢复游戏状态。仅仅将数据放回内存会不会遇到什么问题呢？

当然会。这种方法有很多问题，最主要的问题是由指针引起的，指针的值是一个内存地址，但内存地址在每次游戏运行的过程中是不同的，如果存储了指针的值，下次运行的时候再提取出来继续使用，就很可能会发生冲突，因为它可能会随机的指向内存中的某个位置，这个位置也许还没有被使用，当然根本不可能是所需要的了。

第二个问题是如何恢复这些对象，仅仅将对象的数据存储起来是不够的。否则，当载入数据来恢复游戏的时候，又怎么知道如何处理这些数据呢？如何判断是在为导弹实体还是为镜头实体读取数据呢？无法知道，所以需要一些附加信息来帮助恢复数据。

第三个问题是不能像处理通常的C语言的结构那样处理游戏实体。单单创建它们并为它们赋值是不够的，还需要保证每一个游戏实体都能正常的初始化，那就意味着调用它的构造函数并调用其他的初始化函数。这将使实体有机会将它本身注册到其他管理类中，获得所需资源以及完成其他一些通常要做的工作。

14.1.3 需要完成哪些工作

假定无法通过将数据直接存储到硬盘再从硬盘中重新载入来实现对象的序列化，那么，理想的解决方案是怎样的呢？首先在总体上认识一下要做的工作。然后，再关注对象序列化的特定实现。

希望在任何时候存储任意游戏实体的状态。一般而言，当然希望系统可以正确地处理指针和实体间的关系。如果想要在某一实体销毁时，触发一个任务结束触发器实体，那么尽管它们有可能处在内存中完全不同的两个位置，当游戏载入时，仍然希望触发器和实体

之间可以建立联系。

当想知道需要存储哪些东西的时候，需要区分实例数据和类型数据。类型数据一般由设计人员创建，它包括该类型所有对象的公共特征。例如，一个特定的敌人个体类型可以将最大 hit point（一个 enemy 受到打击，比如刀砍火烧之类后会失血，当血失完了，enemy 也就死了。hit point 就是用来衡量自己可以承受打击程度的一个参数）、特定的武器种类、最高速度和是否具有飞行能力等作为自己的类型数据。如果这些类型数据从不改变，那么就没有必要对它们进行存储。反之，则需要存储，但只存储一次，而且并不是该类型的每个实体都需要存储。敌人个体的实例数据则包括它的当前位置和速度，包括它当前的 hit point，以及武器中剩余的子弹个数，这些都是所要保存的实例数据。

当考虑所要存储的数据的时候，应该认识到存储所有的细节信息是很重要的。游戏需要精确地保存所有细节信息吗？或者说能不能找到一种近似的方法呢？举个例子来说，当保存游戏的时候，一堆火正在壁炉里燃烧，那么很重要的一点要求，就是重新载入游戏的时候，壁炉里的火仍然在燃烧。但是是否需要保证火焰的大小依然相同呢？火焰是否需要恢复到以前的状态呢？大概并不需要。在这种情况下，我们只需要显示一堆火，并表明它已经燃烧了一段时间就足够了。另一方面，保存物体的正确的位置、速度，甚至是加速度则是至关重要的。如果保存游戏的时候，一支箭刚好在空中。那么，毫无疑问，当重新回到游戏的时候，希望它仍然在原来的位置，并且按照已经确定的轨迹运动。

用不同的形式存储和恢复实体数据并不是十分困难，但让解决方案支持这一特性将是非常有用的。这将允许在游戏发售时，使用速度较快的二进制形式，同时也可以使用速度较慢但却更加易于调试和读取的文本形式。总的来说，这是一个几乎可以处理任何形式的序列化的好方法，尤其是在载入效率至关重要的时候。

同时，还希望在任何存储介质上进行实体的序列化。至少可以做到既能在磁盘或内存中对它们进行存取，同时也可以适应其他类型的介质，比如内存卡或是网络管道。当将一个文件调入内存的时候，能够使用内存等不同类型的存储介质是非常方便的，可以根据文件的内容生成相应的实体。

序列化还可以用于在网络间传输实体。尽管应用这项技术看起来很诱人，但是不要指望仅仅通过将网络管道作为传输介质而创造出完整的网络游戏。这种技术可以在其他计算机上创建实体，或者利用这些信息在本机上创建自己明白的新的实体类型，这显然不能满足多人联网游戏实时更新的需要。游戏实体的更新又是一个复杂的问题，需要专门写一本书来讨论。

还没有讨论完全存储和校验点存储之间的区别，校验点存储经常用于游戏终端（游戏机）。它存储在某一关卡某一点的游戏状态，但通常大多数游戏实体都会回到它们的初始状态，只有一少部分状态信息被保存下来，比如玩家的位置和目前所获得的物品等。其他的像门、暗道、敌人、能量等通常会回到它们的初始状态。

校验点存储可以用与游戏实体序列化相同的方法来实现，但只选择其中一部分实体的状态进行存储，当需要将游戏的状态恢复到某一个校验点的时候，可以先退回到关卡的初始状态，然后序列化那些保存的实体。

存盘文件的大小是可保存进度游戏需要关注的一个重要方面。在 PC 机甚至是有着大内存的游戏机上，游戏存盘文件的大小没什么太大的关系。硬盘空间非常便宜，而且存取速度也很快，所以每个文件占用 5~10MB 也是很正常。但是，有时候需要把游戏进度存储到内存卡等一些小型存储器上，所以要存储的文件必须尽可能地小。存储空间就那么大，是多存几个游戏，还是被少数几个游戏存盘文件占满存储空间。关键就在于对游戏存盘文件的大小进行控制，这也是游戏中应用校验点存储系统的主要动机，因为它仅需要很少的存储数据。

14.2 游戏实体序列化的实现

到现在为止，已经列出了游戏序列化过程中所希望达到的目标，接下来的这部分将讨论如何实现它们。

14.2.1 流

流的概念允许抛开具体的序列化存储介质。流是指按照一定顺序排列的一系列二进制数据，这些数据都是可以读写的。此外，也许还需要提供其他的一些操作，比如移动数据到某一特定位置。

标准 C++类库中的 iostream 类提供了一般的输入输出操作，同时还提供了一些模板类，有的支持输入，有的支持输出，还有的两者都支持。可否通过这些已经建好的类来满足序列化的需要呢？这取决于具体情况。大多数时候，这些类比需要的要复杂，需要一些更小巧更灵活的类来满足一般性的需要，以及更好地对各种类型的存储介质进行操作。多数情况下并不需要它们为实现程序本地化和个性化所提供的特性。因此，将直接生成自己的版本；但如果有必要在工程中复用标准 C++类的话，则直接利用它们的接口进行编程。

IStream 接口包含可以使用的所有的流操作。正如在第 10 章中所见，抽象接口允许在运行时转换特定的流的实现而不需要改变其余的代码。这就实现了目标列表中的一项，即对任何类型的媒体介质实现实体的序列化。只要得到该介质的流，所有的问题都可以顺利地解决了。

```
class IStream
{
public:
    virtual ~IStream ( );
    virtual void Reset () = 0;

    virtual int Read ( int bytes, void *pBuffer) = 0;
    virtual int Write ( int bytes, void *pBuffer) = 0;
```

```
    virtual bool SetCurPos ( int pos) = 0;
    virtual int GetCurPos () = 0;
};
```

接下来，可以继承这些类，具体的实现内存流、文件流或所需要的其他类型的媒体介质流。也许还要提供一些特殊平台上的流类型，这些流通过只在该平台上运行的函数来实现，这样就可以在该平台上尽可能高效地运行。例如，一般的文件流可以用 fopen 和 fread 操作实现，但是考虑到减少寻道次数等要求，可以写一个特殊的流来优化 DVD 文件的读取，或者通过使用游戏终端 API 进行低级硬盘访问来优化特定游戏终端上的硬盘读取。

每一个特定的流的实现都有一个构造函数或初始化函数，该函数用适当的参数构造所需的特定的结构。例如，文件流可能会包含一个文件名，或者可能还有一个标识，用来标识以读模式还是写模式打开该文件。内存流可以用一个内存中的具体位置和使用空间的大小来初始化。默认的构造函数没有任何参数，它会在内存中任意位置构造一个流。同样，也可以选择在流类的常规函数中实现这些功能。

```
class StreamFile : public IStream
{
public:
    StreamFile ( const string & filename);
    virtual ~StreamFile();
    // …
};

class StreamMemory : public IStream
{
public:
    StreamMemory ();
    StreamMemory ( void *pBuffer, int size);
    virtual ~StreamMemory ();
    // …
};
```

除了这些基本接口外，也许还需要提供一些辅助函数来增加通用的数据类型的可读性。尽量让接口保持最简单的风格，那些不含任何参数的函数往往更好用。如果更喜欢使用 C++ 流的语法，可以使用“<<”和“>>”运算符来对流进行操作。反之，可以使用 read 和 write 等更直接的函数。

```
int ReadInt ( IStream & stream );
```

```
float ReadFloat (IStream & stream );
string ReadString (IStream & stream );

bool WriteInt (IStream & stream, int n );
bool WriteFloat (IStream & stream, float f );
bool WriteString (IStream & stream, const string & s);
```

还可以实现读写压缩数据的特殊类型的流。这样就把实现压缩和解压缩数据的复杂性转移到流类中。任何其他类型的对象都可以直接使用它。

此外，还可以在流的基础上构造更复杂的文件类型。比如，可以在流的基础上构造分块的二进制文件格式、.ini 文件格式，甚至 XML 文件格式。

下面是一个实现整型数据读写函数的例子：

```
int ReadInt (IStream & stream )
{
   int n = 0;
   stream.Read (sizeof ( int ),(void*) &n);
   return n;
}

bool WriteInt (IStream & stream,int n )
{
   int numWritten = stream.Write ( sizeof ( int ), &n );
   return (numWritten == sizeof ( int ));
}
```

如果需要频繁地写入某些类型的数据，比如指针、向量或矩阵，最好为每一个类型写一个辅助函数。这会使其他的代码更容易实现对它们的序列化。

14.2.2 保存

保存游戏实体的时候，最好的方式是让每个实体自己决定存入流中的最佳方式，为了实现这一目的，需要遍历所有感兴趣需要存储的实体并让它们获得序列化的机会。

1. ISerializable 接口

可以为每个需要序列化的实体调用 Write 函数。可以让该函数成为 GameEntity 基类的一部分，这样所有的问题都解决了。对于那些不需要序列化的实体可以让它保留为空函数，而其他的实体则可以根据自己的需要去实现它。

一个更好的方法是让与序列化相关的函数成为抽象接口——ISerializable 接口。这样

GameEntity 基类就可以继承这一接口，然后再按照上面的方式进行。然而，让这些函数分化为各个独立的接口可以更易于对其他类型的对象进行序列化，而这些对象不一定非要是游戏实体类型的。如果确实需要知道一个对象是否实现了 ISerializable 接口，就要使用第 10 章中所讲的 QueryInterface 方法。现在假设所有的游戏实体都实现了 ISerializable 接口。

那么，如何实现 ISerializable 接口呢？其实很简单，看看下面这些代码就知道了：

```
class ISerializable
{
public:
    virtual ~ISerializable () { };
    virtual bool Write ( IStream & stream ) const = 0;
    virtual bool Read ( IStream & stream ) = 0;
};
```

显然，还可以在载入时使用 ISerializable 接口，这就是 ISerializable 接口中有 Read 函数的原因。

如果更喜欢用 C++流的语法结构，可以使用“>>”和“<<”运算符。反之，如果想更加别致一点，可以写两个函数来调用 Serialize，但是这两个中必须有一个是常值函数（const 类型的函数），该函数仅能写。而另一个是非 const 类型的函数，非常值函数可以进行读取。但这样会带来更大的麻烦，它使得不能区分正在调用的是哪个函数，使用简洁的接口要保证不影响对程序的清晰简洁。

2. 实现写入

为每个实体实现 Write 函数是非常简单的工作。要决定需要存储哪些数据，然后将每个想要存储的成员变量序列化到流中。对于整型、实型或其他一些标准的数据类型，直接将它们写入流中就可以了。但如何处理那些本身具有成员变量的实体呢？通常必须确保这些成员变量也实现了 ISerializable 接口。这样，只需要调用它的 Write 函数，该函数用相同的方法序列化每一个成员变量。利用这种方法可以毫不费力地存储任意多个嵌套对象。如果一个实体除了它本身的内容外还包含其他对象的指针或引用，则需要用不同的方法进行处理（这种方法留到下一部分中讨论）。

如果在实体类中用到继承关系，也许想让父类对它自己的数据进行序列化处理。这样派生类只需要关注它们新添加的变量即可。

下面是镜头类的 Write 函数的一些示例代码：

```
bool GameCamera :: Write ( IStream & stream ) const
{
    // Let the parent class write common things like position ,
    // rotation , etc.
```

```
    bool bSuccess = GameEntity :: Write (stream);

    // these are basic data types, serialize them directly
    bSuccess &= WriteFloat ( stream m_FOV);
    bSuccess &= WriteFloat ( stream m_NearPlane);
    bSuccess &= WriteFloat ( stream m_FarPlane);

    //this is an object that needs to be serialized in turn
    bSuccess &= m_lens.Write ( stream );
    return bSuccess;
}
```

到现在为止，必须实现的是纯二进制形式，没有头和附加信息，只有原始数据。当要求实体尽可能快的载入的时候，比如在已发布的游戏中，这是一种很好的格式。但这种格式对游戏的开发并不是很有利，只要实体类发生一点小小的改动，所有以前保存的游戏信息都无法继续使用。更糟的是，没有方法检测这类问题是否正确的话，很可能会因此而读到垃圾数据。

鉴于以上原因，一个理想的方法是同时实现两种以上的形式，一种是上面看到的较快的这种，另一种则更易于调试，当格式发生变化时程序仍能正常工作，甚至有可能会选择基于文本的格式，以便在游戏开发过程中更方便地进行调试和检测。

3. 惟一标识

必须要解决的另外一个问题是如何存储指针，对此，有几种选择。

第一种可能的方法是尽量避免使用指针或者至少避免使用指向实体的指针。可以用代表游戏实体的惟一标识 ID（或 UID）来代替指针。如果每一个实体都有一个能够保证决不重复的惟一 ID，这样就只需要保存这个标识。当想要直接处理实体的时候，只要通过游戏实体系统将与该标识相对应的实体指针传给我们就可以了。

除了解决指针存储问题以外，这种方法还简化了游戏实体间的联系。如果在实体中直接使用指针，那么当一个实体在游戏中被删除的时候，另一个实体却试图对它进行调用，这时将会发生什么情况呢？理论上，可以不删除任何实体而让它们一直存在下去，只要保证不对它们进行更新和渲染。如果使用 UID 方法，即使这个实体已经不存在了，也只会得到一个空指针，并且知道不能再对它进行操作。

例如，下面这段代码使用 UID 方法更新了已锁定目标的自动跟踪发射装置的位置。

```
void HomingProjectile :: Update ()
{
    if ( !m_bLocked)
        return;
```

```
    GameEntity *pTarget = GetEntityFromUID (m_targetUID);
    if ( pTarget == NULL) {
        m_bLocked = false;
        return;
    }

    // Do whatever course correction is necessary here ...
    //...
}
```

这与在 13 章中为解决共享对象问题而使用的方法非常类似。关于如何生成句柄，说明和 13 章里的一样，并且也应用了同样的从句柄到指针的转换办法。

4. 资源

那些指向资源而不是实体的指针又该如何处理呢？通常这不是什么太大的问题。让实体引用资源，因为它们从一开始就是这样创建的，数据也都是这样组织的。玩家扮演的那个实体的属性之一是渲染用的网格以及它的动画和纹理。通常实体通过文件名或资源 ID 来引用资源，而这正是所需要的。如果在程序运行期间它所指向的资源发生了变化，实体应该将重新生成的文件名或 ID 号保存下来已备将来恢复之用。反之，如果它始终保持不变的话，就没有保存的必要了。

5. 保存指针

在实体之间使用指针一直是有争议的问题，它可以使代码编写更加容易。但需要处理游戏中实体销毁的问题，如果不考虑这一点的话，对所有的实体都使用 UID 就有点得不偿失了。

有可能游戏实体在开发过程中并没有考虑序列化问题，对游戏的保存被留到游戏开发周期的最后阶段才实现。或者有意把它留到第一版开发完成以后，我们打算在重用这些源代码开发游戏的续集的时候加入序列化功能。

在这种情况下，将已有代码的指针转化为 UID 是一个重要的工作。试想一下，在成百上千个类中将指针及相关的代码转化为用 UID，是怎样一种可怕的情况。如果只是想要一种更快的序列化实体的方法的话，更好的选择是把指针直接保存到磁盘。

当重新载入以后，指针中的内存地址已经不能指向正确的位置了。显然，当载入游戏实体的时候，需要采取一些措施来解决这个问题。现在，我们要做的是保存原始指针，先不要管它，下一部分会讲述如何对它进行进一步的处理。

14.2.3　载入

到现在为止，所做的一切都只是为载入游戏实体和重置游戏状态做准备。下面将对游

戏载入本身进行讨论。

1. 创建对象

在恢复各种不同类型的实体的过程中，必须要做的就是根据从流中读出的数据创建实体类型。仅仅能够读出 GameCamera 类的数据是不够的，需要知道哪些属于 GameCamera 类，并且需要创建一个该类型的对象。

如果觉得这一问题听起来比较熟悉的话，那是因为已经在第 13 章中读到它了。只要给出类名或是类型标识，一个好的游戏实体工厂应该可以创建任何实体类型。那么就可以对刚刚创建的实体调用 Read 函数来从流中载入数据。

```
string strClassName = ReadString ( stream );
GameEntity * pEntity = EntityFactory :: Create ( strClassName );
// … Some bookkeeping here …
pEntity->Read ( stream );
```

根据工厂系统类型的不同，可以保存实体类名的全部字符串，并将该字符串传到工厂系统来创建它们。使用字符串之前通常需要权衡利弊：字符串很容易调试并且可读性强，但它比较慢而且占用内存比简单标识要多。尽管 32 位的标识符（指 UID）效率较高，但在调试器中查看它们的时候，不能从这些标识符中很快地看出所创建的对象的实际类型。

2. 载入指针

怎样处理指针的这些棘手的问题呢？正如前面所提到的，可以直接将它们保存在代码中，并让它们得以正确的恢复，这就是解决这一问题的方法。

我们知道每个内存地址是惟一的。通过存储一个实体的内存地址可以惟一的标识它。如果存储它的时候连同它的指针一起存储，就可以构造一个对照表，这个对照表允许在载入时将旧的地址换成新的地址。

为了保证转换工作的正常进行，对照表需要在实体载入时就已经生成。否则，可能会去查找一个还没有载入的内存地址。载入的过程是这样的：首先载入所有实体并构造一个新旧地址间的映射表；然后修复性遍历所有的实体，使所有指针都有机会得到修复并保证它们指向最新的正确的地址。

为了实现对指针的修复，需要从载入系统和 ISerializable 接口中得到更多的支持。当载入所有实体以后，将让所有实体有机会修复它们的指针所指向的地址。为了实现这一目标，扩展 ISerializable 接口来增加一个 Fixup 函数。

```
class ISerializable
{
public:
```

```
        virtual ~ISerializable () { };
        virtual bool Write ( IStream & stream ) const = 0;
        virtual bool Read ( IStream & stream ) = 0;
        virtual void Fixup () = 0;
};
```

与实现实体本身的 Write 和 Read 函数的方法相同，可以构造一个 Fixup 函数来负责将每一个已保存的指针从旧的地址转化为新地址。如果一个实体没有保存指针，则不需要实现 Fixup 函数，这时 GameEntity 基类会自动生成一个空函数。像其他的序列化函数一样，一个实体除了对它自身的指针进行转化外，还必须调用它父类的 Fixup 函数。

为了实现修复过程，每一个实体需要在被重新载入内存的时候与它的旧地址相关联，要解决这一问题，可以在实体写入流时就将它们的指针保存起来。

上面这些信息为正确处理指针做好了准备。每当从流中创建一个实体的时候，也要读取旧地址的内容，并将它们与新地址一起放入对照表中。AddressTranslator 类负责确定所有的地址，并在修复过程中转换它们。

```
GameEntity * LoadEntity ( IStream & stream )
{
    string strClassName = ReadString ( stream );
    GameEntity * pEntity = EntityFactory :: Create ( strClassName );

    void *pOldAddress = ( void* ) ReadInt (stream);
    AddressTranslator :: AddAddress ( pOldAddress , pEntity );

    pEntity->Read ( stream );
    return pEntity;
}
```

AddressTranslator 类的 AddAddress 函数将新地址按照旧地址的索引顺序存入一张哈希表中，这样从旧地址到新地址的转换效率会很高。

为了实现 Fixup 函数，需要用到 AddressTranslator 类的另一个函数——TranslateAddress 函数。这个函数从哈希表中查找旧的地址并取出新地址。下面是 HomingProjectile 类的 Fixup 函数：

```
void HomingProjectile :: Fixup ()
{
    m_pTarget = (GameEntity * )AddressTranslator:
```

```
                                        :TranslateAddress( m_pTarget );
}
```

完成载入并将所有的指针都修复以后，由于哈希表将不会再次被用到，所以应该销毁哈希表以节约内存。

需要注意的是，这种方法只对那些已经保存并且添加到表中的指针才有用。在这种情况下，以上过程对所有的游戏实体都适用。如果试图对没有添加到对照表的指针进行操作，系统应该弹出一个警告，通知转换过程发生错误。否则，问题将被隐藏起来，由此而留下来的 BUG 只有经过彻底的检查之后才有可能被查出来。

尽管是很细微的问题，仍然需要小心地存储对象的内存地址。这个地址必须是被其他游戏实体所指向的正确地址。哪怕是最微小的偏差都会导致转换过程的失败。什么情况会导致这种微小的偏差呢？多重继承有可能会产生偏差。当使用多重继承的时候（见第 2 章），对不同的父类进行强制转换，会导致指针本身的值产生偏差。幸运的是，在这种情况下，GameEntity 继承了 IRenderable 接口，所以不需要担心什么。如果它继承了另一个抽象接口或使用独立的基类，那么就必须十分小心的对 GameEntity 类型的指针进行及时的保存。

14.3 组装起来

光盘中的序列化实例包含了存储和恢复游戏状态完整的程序代码。Visual Studio 工程的工作区在 Chapter 14. Serialization\ Serialization.dsw 中，它创建了一个简单的 GameEntity 对象树，在硬盘中对它进行存储、删除，再重新载入。它分别输出了保存之前和从硬盘中恢复以后的树的内容。以此来证明程序的运行过程与所期望的相同。

这个程序只作为应用本章中概念的一个运行实例。它并不能作为健壮的代码添加到任何开发环境中去，它没有很好的处理错误，并且文件的格式也都是很原始、很基本的。通过让代码保持简单精练的风格，可以更容易地理解它所体现的概念，而不是拘泥于复杂的层次和出错处理。例如，游戏实体工厂只是使用硬编码实体类型的一个函数。它是可以运行的最简单的解决方案，但可以在第 13 章中更好地理解它的实现方法。

需要特别注意的是，所有实体的所有指针是如何被正确地存储和恢复的。所有的实体都有一个指向它的子实体的指针数组，这些指针只是简单的被存储，然后在修复过程中被转换。此外，镜头实体拥有一个指向其他游戏实体的指针，用于指向它所聚焦的实体。这个指针也被用相同的方法处理并被正确地恢复。图 14.1 展示了光盘的示例代码（\Chapter14.Serialization\ Serialization.dsw）中被序列化的游戏实体树的结构。

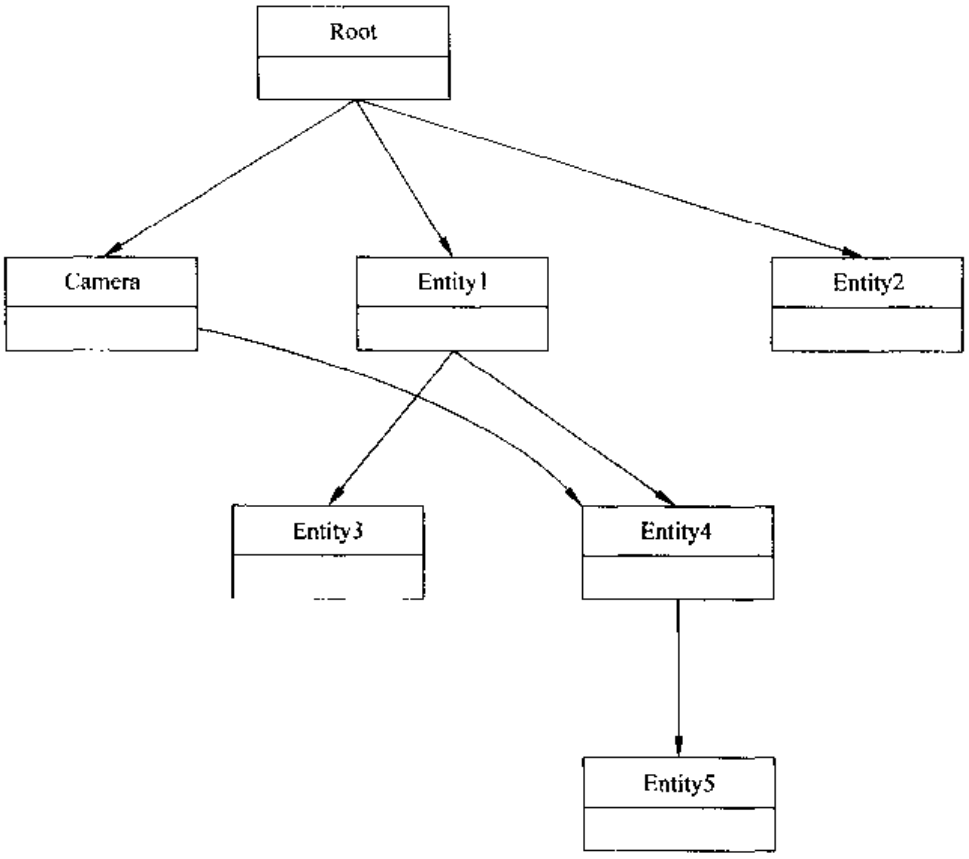


图 14.1　示例代码中序列化的简单实体树

14.4　结　论

本章中介绍了像游戏进度存储这样看起来很简单的功能，要想正确地实现，也是有一定难度的。游戏存储只是序列化问题的一个例子：将游戏的状态存储到一些数据介质中，并在以后完全恢复。

其他语言提供了更好的内置序列化工具，但是在 C++中，必须采用一种麻烦的方式自己实现它。特别地，当处理复杂对象结构的时候，需要处理保存和恢复内存指针的问题。

本章介绍了序列化的几种不同的形式。它们全部依赖于流的使用，这使不必考虑序列化实体所用的特定的介质，可以提供对硬盘文件、内存甚至是网络进行操作的流。

可以看出游戏实体的状态是如何使用 ISerializable 接口来存储的，即让每个实体自己选择要存储的内容。还讨论了使用 UID 代替指针建立实体间关系模型的可能性。如果程序并非从一开始就采用这种方法的话，转换过程将是一项非常耗时的工作。

最后讨论了使用流来恢复对象的方法。还学习了如何对旧的内存指针进行转换，以保证它们在游戏实体被重新载入后仍能正常工作。

14.5 阅读建议

与使用最新的图形技术实现惊人的效果相比，游戏进度存储不太具有吸引力。所以很难找到这方面的太多著作，下面这些书中有一些关于在游戏中使用对象序列化技术的简短而有趣的章节：

Eberly , Daid H., 3D Game Engine Design ,Morgan Kaufmann ,2000.

Brownlow, Marin, "Save Me Now!" ,Game Programming Gems 3,Charles River Media , 2002.

最初的工厂模式是在下面这套书（总共4本）中介绍的：

Gamma ,Erich ,et al., Design Patterns,Addison-Wesley ,1995.

第 15 章　处理大型项目

本章将讲述：

- 逻辑结构与物理结构
- 类和文件
- 头文件
- 库
- 配置

到现在为止，讨论的都是一些抽象概念：内存分配、插件或是运行期类型信息，这些知识对于小型项目或快速开发实例是足够的。但在开发完整的游戏项目时，这些知识还远远不够。当游戏的全部编译过程超过一个小时，或者发布一个备选版本的过程麻烦而又容易出错的时候，即使最好的数据结构或是最高明的算法也无济于事。

本章主要讨论大型游戏项目中代码库的处理。并将弄清楚为什么应该尽可能避免文件之间过多的联系，同时介绍一些改善这种情况的规则，这些规则同时还能加速编译的过程。还会讨论如何让工程更容易地为不同平台生成不同的版本的问题。

学完本章以后会获得足够的技巧，从而能够自信地应对一个大型的游戏项目。

15.1　逻辑结构与物理结构

程序的逻辑结构是大多数 C++ 书里都会讲到的主题，它涉及类、算法、数据及它们之间的关系。显然，逻辑结构对程序的成功是非常关键的，数据结构的选择决定了程序的执行效率，类之间的关系决定着维护和调试的复杂程度。

而程序的物理结构则处理一些更加具体的内容，例如文件、路径、工程或 makefile，这是一个比较枯燥的问题，并且没有得到应有的重视。但随着项目规模的增大，它已经变得同逻辑结构一样重要了。完全相同的程序可以由许多不同的物理结构实现，但不是所有的物理结构都能够实现较快的编译速度，更容易地扩展成可以构建多个不同版本；很多物理结构只会导致漫长而痛苦的编译链接过程，生成不同版本时会遇到很多的麻烦。

为了更好地了解这些，看一些真实的数据。一个普通的 PC 或游戏终端游戏项目可以使用代码库，这些库由 4000~5000 个文件所组成。更别提那些玩家众多的网络游戏了，这些网络游戏的代码库加上服务器代码、计费、备份等，通常是普通游戏的好几倍。对于这种规模的游戏，物理结构是至关重要的。一个结构良好的工程通常的编译时间是 5~10 分钟，

并且小的改动几乎不需要重新编译。相反，一个结构较差的工程的编译时间会轻易地超过一个小时，而且一个小小的改动都会导致重新编译 10~20 分钟。在这两者之间选择哪一个呢？

物理结构是由文件在编译过程中对其他文件的依赖情况决定的。在 C++里，需要另一个文件就意味着使用#include 预编译命令。在理想情况下，每一个文件只编译它本身，但这显然是不可能的，因为一个程序是由很多个交互的对象组成的，所以它们需要彼此有一定的联系。实现文件之间的联系最小化是一个可以接受的物理结构所要达到的最终目标，这种文件与其他源代码之间的关联程度称为绝缘性，文件间的联系越少其绝缘程度就越高。

幸运的是，一个好的物理结构总是与好的逻辑结构紧密相连。一个类对外隐藏了它的代码称为封装良好。一个类的封装性越好，其他类与它的联系就越少，逻辑结构也就越清楚，通常会带来如下优点：维护容易，调试清晰，测试简单。

一般的规则是，每当一个文件被修改时，其他包含它的文件也会随之发生改变并需要被重新编译。理想情况下，对一个文件的改动只会引起一两个文件的重新编译。一个非常混乱的物理结构会因为一些看起来不相干的变化而导致工程中大多数文件的重新编译。从修改到运行游戏，再到测试结果之间的时间间隔很快会变得让人无法忍受，那将导致工作质量降低，特征测试（Feature Test）很难进行。为了避免较多的重新编译不得不对代码进行一些特别处理。

本章的其余部分将着眼于物理结构的方方面面，将讲解实现好的物理结构的可能的组织形式和解决方案，以及如何在大型的工程环境里尽可能提高程序员的工作效率。

15.2 类和文件

文件是一个程序的物理结构的核心，正确地组织文件是实现高效的物理结构的第一步。通过浏览已有的 C++代码可以找到许多有用的信息；其中一些不太常见，因此可能会带来一些意外的惊喜。通过对这一部分的学习将对为什么要将类分割成不同的文件以及怎样将其分割成不同文件有一个深刻的理解。

总的来说，一个应该遵守的较好的规则就是将每个 C++的类分成两个文件：一个头文件（通常的扩展名为.h 或.hpp）和一个实现文件（扩展名为.cpp）。这种组织形式有几个优点。首先，它便于对文件进行浏览，并且很容易通过文件名找到某一个特定的类。甚至在集成开发环境（IDE）或是其他代码浏览工具中，这种简单的系统是很有用的。

其次，它很好地处理了缩短编译时间，特别是那些额外的编译时间的问题，这也是最关心的问题之一。因为当编写新的代码或是修补 BUG 的时候，每天都要编译许多次。如果将许多类压缩到一个文件，编译的速度会更快，但那样就不得不为一个类的改动而编译更多的代码。将一个类分成更多的文件也不会带来额外的好处，因为当发生改动的时候，很可能整个类都必须重新编译。

再次（这是我个人比较主观的观点），将整个类在一个文件中实现看起来像是最好的

办法。一个文件最好是不要太长以至于无法组织，甚至让我们感到无从下手，但它也不应该太短以至于失去了意义。如果一个文件大到让你想把它分成多个文件，最好将它分成多个类来实现。

15.3　头　文　件

头文件是外界观察类的窗口。它的信息主要是让其他的代码了解该类以及它的使用方法，除此之外，再没有其他的信息。其他的代码通过直接在它本身的文件中加入#include 语句来使用头文件。

15.3.1　哪些内容应加到头文件中

那么哪些内容应加到头文件中呢？简单的答案就是头文件里应包含允许程序正常编译和运行的最少代码。这一点说起来容易但做起来很难，特别是当具有 C 语言背景时，在 C 语言里头文件常常包含除了实现代码以外的所有东西。预处理#include 语句对程序的结构是非常重要的，需要作一个单独的部分去讨论它，所以它将是下一部分的主题。现在对头文件的讨论中先将其忽略。

第一步，先将那些非必要的信息从头文件中移到实现文件中。那些只在实现文件中用到的常量、小的私有结构或者类都是要移出头文件的对象。要经常问自己这是与其他类进行交互所必须的吗？

不幸的是，C++强制将一些本应放在实现文件里的内容放到头文件中，从而让情况变得糟糕。一个类的私有部分只有类本身对它感兴趣，但却被强制在头文件中声明私有函数和私有变量。这将产生一个不好的结果，就是会在头文件中暴露一些多余的信息。在下一节中会介绍一些关于这一问题的可能的解决方案。

在生成头文件时，还需要考虑如何让它们在程序的其余部分使用。头文件的使用非常简单，它们被包含到所有使用该类的地方，这通常不是什么问题。只有一少部分文件包含了特定的头文件。但是有时候一些特殊的类似乎在任何地方都需要：包含程序用到的错误处理代码、资源文件或是包含许多 ID 转换表的头文件。

到处包含头文件是很危险的，如果它们几乎不被改动的话还可以容忍。然而，如果它们可能在程序开发过程中发生改动，则应该尽可能避免。一个包含所有全局错误代码的文件可能被到处包含，并且在添加新的错误代码的时候频繁地更新。比如，决定加入新的错误代码来指出一个网络用户已经退出。把它加入到头文件，选择快速编译来检验更改，然而会惊奇地发现它触发了整个工程的重建。每个类库、每个 DLL 以及每个单独的源代码文件都需要重新编译。在大型工程中这可能要花上几个小时的时间。

对于这种文件，最好将它分解为几个不相关联的文件。每个文件包含了不同子系统所用到的错误代码。图形渲染可以作为其中的一个，此外还有网络代码、文件 IO，添加新的

网络错误代码只会导致网络子系统及其所有链接的重新编译，并将修改所需要的时间减少到5分钟。

15.3.2 包含控制

#include 是预编译指令。这意味着它将被预处理程序解释，后者可以在编译之前对源文件进行改动。这时，预处理程序在每个单独的文件中扫描#include "filename"指令，打开所指定的文件，读取其中的内容，并把它们插入到#include 语句所在的地方。

事实上没有只能对头文件使用#include 指令的规定，可以对任何文件使用它：头文件、实现文件、文件的一部分或是任意的文本文件。然而，传统上，#include 指令只用于包含头文件，这些头文件通常包括类的声明、函数的声明、全局变量以及宏定义，尽管有时也会看到其他的不同的用途。

看下面这段代码：

```
// Game.h
#define MAX_PLAYERS 16

// Game.cpp
#include "consts.h"
Players players[MAX_PLAYERS];
```

被预处理程序转换成如下形式：

```
// Game.cpp
#define MAX_PLAYERS 16
Players players[MAX_PLAYERS];
```

这些正是编译器所能看到并进行处理的部分。

使用#include 指令会导致一些潜在的问题，这些问题在开始的时候不是很明显的。考虑一下下面这个多余的头文件，它是对上面这个例子的改动。

```
// FrontEnd.h
#include "Game.h"
// Rest of the front end declarations…

// Game.cpp
#include "Consts.h"
#include "FontEnd.h"
```

```
Players players[MAX_PLAYERS];
```

只看到 Game.h 没人会认为它有什么错误，然而试着去编译它就会产生一个编译错误。这时，编译器会指出我们试图对同一个常量定义两次。

问题在于 Game.h 的内容在同一个文件中被包含了两次。下面就是预处理程序处理后的程序：

```
// Game.cpp
#define MAX_PLAYERS 16
// Rest of the front end declarations...
#define MAX_PLAYERS 16
Players players[MAX_PLAYERS];
```

现在可以非常明显的看出正试图对常量 MAX_PLAYERS 定义两次。在没有意识到的情况下，编译器对同一个文件包含了两次。这不仅浪费了编译时间，而且导致了编译错误。

为了解决这个问题需要使用包含控制器。包含控制器是防止在同一个编译单元中对同一个文件包含一次以上的预处理指令。包含控制器广泛的用于 C 和 C++程序中，它通常以下面这种形式实现：

```
// Game.h
#ifndef GAME_H_
#define GAME_H_

#define MAX_PLAYERS 16

#endif  // #ifndef GAME_H_
```

如果每个头文件里都加入对包含的控制，就可以保证它不会被包含两次。预处理程序第一次包含它时，控制符号（这里是 GAME_H_，但其他头文件可能不同）还没有被定义，它立刻被定义并包含了文件的其余部分。当试图第二次包含同一个文件时，控制符号已经被定义，预处理程序会完全跳过这个文件。

控制符号的选择并不是很重要，除非有一个很大的代码库以至于有可能在同一个编译单元中使用两个相同的控制符号。通常，用头文件名的变形是最简单的解决方案，并且对大多数工程这种方法都运行良好。

预编译指令#pragma once 的使用是在查看源代码时，有可能遇到的一项相关技术。这条指令防止预处理程序在同一编译单元中重复包含（甚至重复打开）含有#pragma once 的文件。因此它看起来，在防止文件被多次包含方面与那个麻烦的包含控制器具有相同的功能，而且还能防止文件被第二次打开，这一点在一定程度上提高了编译的速度。

实践证明，#pragma once 并不是包含控制器的特别好的代替品。它的主要优点是编写清楚、简单，但也仅此而已。它的主要缺点是它并不是标准 C++的一部分。很多的编译器可以正确地识别并应用它，但并不是全部。如果做的是跨平台开发，那么就得认真考虑这个问题。

经常看到的是在同一个文件中同时用到它们两个，包含控制器用于防止文件在不支持#pragma once 的编译器中被重复包含，而#pragma once 在支持它的编译器中避免文件被多次打开。由于并不是所有的编译器都支持#pragma once 指令，所以它通常被包含在条件编译语句中，从而只允许支持它的编译器进行处理。典型的头文件书写格式如下：

```
#ifndef MYHEADFILE_H_
#define MYHEADFILE_H_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

// Contents of the header file here
// …
#endif  // MYHEADFILE_H_
```

在这里，_MSC_VER > 100 意味着，只希望对某一版本以后的微软编译器使用#pragma once。当然这不会减少代码量或者让程序更整洁清晰。尽管不得不在每次增加新的支持#pragma once 的编译器的时候修改所有的头文件，但如果有助于减少编译时间，也是可以接受的。

需要指出的是，多数情况下，#pragma once 都是不必要的。大多数能够识别#pragma once 指令的编译器都具有识别包含控制，并自动地避免多次打开该文件的能力。因此最后，只在头文件中添加包含控制是最好的解决方案。

如果正在试着调试一个可能由预处理引起的问题，那么可能需要求助于编译器文档。多数编译器都有一个很方便的特征，它们可以在编译过程完成以后按照自己的方式将源代码保存起来，所以可以查看编译器处理和编译的过程。这将更易于调试宏代换（#define）或者检测任何奇怪的或者不正确的文件包含。

一些编译器还提供了另一个有用的功能，它有一个开关可以让编译器将每个包含的文件打印出来，这不仅可以追踪包含错误，而且你会发现，那些看起来单纯的类自身可能已经包含了数以百计的头文件。

15.3.3 .cpp 文件中的包含指令

#include 指令完全决定了程序的结构，你的代码是简单高效，还是很复杂，像一个难

看的怪兽将长长的触角伸向每一个单独的文件，这完全取决于#include 指令的使用情况。

让每个实现文件里的包含的头文件的数量达到最少，这不仅指.cpp 文件中的#include 语句，而且还有那些相关头文件中的#include 语句。在一些极端的情况下，编译一个.cpp 文件会包含数以百计的头文件。光是访问和打开这些文件所要花费的时间就是非常可观的（每个.cpp 文件要一两分钟的时间），再乘以工程中.cpp 文件的个数，那么半个小时或更长的编译时间会让我们感到很突然。

减少包含文件的数量和最终的编译时间的第一步是要解开#include 造成的盘根错节，并用一种可以弄清楚的方式组织它们。

通常，.cpp 文件在文件的顶部包含了一些头文件。文件的数量从一两个到几十个不等。每一个#include 指令将它所指向的文件的内容全部添加到它所在的位置。那么文件本身和所有#include 语句扩充到它里面的内容都会照常编译。

一个必须要注意的问题是，#include 是按照它们在代码中出现的顺序来展开的，编译过程也是从上到下进行的。所以当第二个头文件的内容被编译的时候，第一个头文件的内容已经被识别，但它后面的其他头文件的内容还没有。这就意味着如果不重视包含的组织，同一个头文件有可能在某一个.cpp 文件里运行良好，但在另一个文件中却编译失败，原因只是由于#include 指令前面的内容不同。

可以通过遵循一个非常简单的规则来解决这个看起来复杂的问题：在一个包含某一个类的实现代码的.cpp 文件中，第一个包含的必须是对应于这个类的头文件。后面处理预编译头文件时会讲到一个特例，但除此而外应该把它当做一个通用的规则。

根据这一规则，代码应该与下面的形式类似：

```
// MyClassA.h
class A
{
   // …
};

// MyClassA.cpp
#include "MyClassA.h"   // <- First include is the header file for the class
#include "MyClassC.h"
#include "SomethingElse.h"

// The rest of MyClassA implementation
// …
```

究竟要达到什么样的目标呢？就是要避免让依赖于其他头文件的头文件过早的出现。即使它是包含列表里的第一个，也能保证它的正常编译。通过让所有的类遵循这一规则，其余头文件的顺序也都不会存在任何问题。

这个规则所导致的一个结果是：一些头文件将不得不包含其他的头文件以便能够正常的编译，这是一个很好的做法。如果它们需要，可以在头文件中加入#include 语句，而不需要记住在使用该头文件的.cpp 文件中包含它们。

15.3.4 头文件中的包含指令

在头文件中最先应该被删除的就是那些不必要的#include 语句。这听起来有点怪，但不幸的是，让一些不需要的#include 语句混杂在头文件中是非常普遍的。比如，一些头文件可能在某些时候被包含进来，随后代码发生了改动，我们不再需要它们了，但很少有人记得去删除这些#include 语句。无论是否需要，都会在每次编译程序的时候为它们付出代价，所以还是把它们删掉吧。只有那些缺少它们就会无法编译的#include 语句才放在头文件中。

删除了那些没有用的包含语句以后，可以把注意力转移到必须用到，但可以被其他方法替代的包含上来。在头文件中包含文件的惟一原因就是，头文件中要用到某些在包含文件中声明的内容。例如，当从一个类派生另一个类时，必须包含父类的头文件。

```
// B.h
#include "A.h"

// B inherits from A , so we must include A here.
// There is no way around it
Class B : public A
{
// …
};
```

继承是必须包含父类的头文件的一种情况。一般没有避免它的方法，因为编译器需要看到父类的全部内容才能知道怎样正确地构造派生类。

但是，有时候编译器并不需要看到一个类的全部声明，它所需要知道的仅是类的名字，这种情况发生在使用一个类的指针或引用的时候。只要将类名告知编译器，不需要包含类的声明，只要使用一个类名的前向声明就可以了。考虑一下下面这两个文件描述的 GameCamera 类：

```
// GameCamera.h
#include "GameEntity.h"
#include "GamePlayer.h"

class GameCamera : public GameEntity
{
```

```
    GamePlayer * GetPlayer();
  private:
    GamePlayer * m_pPlayer;
};

// GameCamera.cpp
#include "GameCamera.h"

// Rest of GameCamera implementation…
```

初看起来，GameCamera.h 文件中的这两个#include 语句似乎是必须的。当然，第一个是需要的，因为 GameEntity 是父类。但是，包含 GamePlayer.h 的惟一原因就是需要用一些指向该类对象的指针作为成员变量。鉴于它们只是一些指针，可以使用前向声明来避免包含文件的所有内容。

```
// GameCamera.h
#include "GameEntity.h"
class GamePlayer;  // Forward declaration is enough

class GameCamera : public GameEntity
{
    GamePlayer * GetPlayer();
  private:
    GamePlayer * m_pPlayer;
};

// GameCamera.cpp
#include "GameCamera.h"
#include "GamePlayer.h"

// Rest of GameCamera implementation…
```

注意当从头文件中删除#include 语句时，必须在实现文件中添加它们。这是因为 GameCamera 在实现代码的某些地方用到了 GamePlayer，这就需要让编译器看到 GamePlayer 类的声明。

注意，本可以将#include 语句保存在头文件中，但却将它移到了实现文件中。这样做有什么好处呢？事实上，好处是显而易见。假设有 100 个其他的类需要包含 GameCamera.h。通过将对 GamePlayer.h 的包含从 GameCamera.h 文件移到 GameCamera.cpp 中，可以避免在编译过程中将 GamePlayer.h 文件打开 100 次。如果 GamePlayer.h 文件中还分别包含 10 个其他文件，那么就省去了对 1000 个文件的读取！现在只在 GameCamera.cpp 文件中包含

GamePlayer.h 文件，这使我们只对它编译一次。试想一下在上千个文件组成的工程中，类似这样的例子到处都是，那就可以看到，适当的使用前向声明会对一个大型工程的物理结构和编译时间产生多大的影响。

15.3.5 预编译的头文件

即使所有的代码都遵循了本章中提到的包含规则，还应注意到有多少相同的头文件被包含了多次。因为大部分代码都使用 C 的标准类库头文件、STL、Windows、OpenGL 或其他一些通用 API。因此几乎所有的文件都包含一些这样的头文件。

```
// MyClass.cpp
#include "MyClass.h"
#include <vector.h>
#include <list.h>
#include <algorithm.h>
#include <stdio >
#include <windows.h>

// Now we can start including the other files from our code
#include "MyClass2.h"
#include "MyClass3.h"
```

情况比它看起来要更糟糕的多，因为这些大型 API 拥有大量复杂的头文件，而这些头文件又各自包含了许多其他的文件。最大的问题是这些头文件在开发过程中是不会发生变化的，每次编译时都需要包含和处理它们。在这种情况下编译速度会非常缓慢。

幸运的是，一些编译器提供了解决方案：预编译头文件。由于预编译头文件是一种只在一些特定的编译器上可用的选择，所以不能在跨平台上的项目里使用它。它是一种可靠而又实用的方法，值得在所有可能的时候使用。如果对特定的编译器不太确定，可以查看它的文档并出找出使用它的方法。

究竟什么是预编译头文件呢？从它的名字中就可以看出来。使用它的时候，一些文件只被编译一次，并且编译的结果保存在磁盘中以便以后在其余的程序或后续编译中继续使用。这就意味着所有被这些文件包含的用于创建预编译头文件的头文件将只被编译和处理一次。

此外，每次编译程序的时候，只要生成预编译头文件的文件没有被修改，这个预编译头文件就可以不经编译而直接使用。这对那些几乎在程序的每个文件中都用到大型外部 API 的头文件是非常理想的选择。它们只被编译一次，以后就可以在编译过程中随意被使用。

不同的编译器对文件中使用预编译头文件有不同的规定。在微软的编译器中，必须确保：特定的头文件被作为第一个包含文件包含在.cpp 文件中。可以查看编译器文档以了解

细节内容。在包含规则中，这是惟一的一次例外。除此以外，一个类的.cpp 文件的第一个包含语句必须是这个类的头文件。一个使用预编译头文件的.cpp 文件通常具有类似下面的形式：

```
// MyClassA.cpp
#include "precomp.h"      // Precompiled header files
#include "MyClassA.h"  // First real include is the header file for the class
#include "MyClassC.h"
#include "SomethingElse.h"
// The rest of MyClassA implementation
// ...
```

预编译头文件的效果是令人吃惊的。所以如果还没有准备使用它，研究一下它在工程中的应用还是值得的。在一个工程中使用预编译头文件，有可能会使它的编译速度提高 10 倍。这取决于工程的结构和它使用其他大型头文件的多少。在一些非极端的情况下，仍然有希望使用预编译头文件将编译速度提高两到三倍。

预编译头文件的主要缺点就在于，它对头文件的包含的随意性会引起所包含的文件的数目随着项目的进行不断的增加。这就意味着每一个使用预编译头文件的文件（多数情况也都是如此）都会自动获得预编译头文件的所有组成文件。这本身并不是什么坏事，因为这并不增加编译时间，但它会将这些头文件暴露给那些从来不需要使用它的源文件。

例如，将 windows.h 添加到预编译头文件后，将导致 Windows 的结构、常量和函数暴露给每一个使用预编译头文件。迟早会发现，这些 Windows 的函数或数据类型会遍布程序各处。这确实是一个问题。如果正在做多平台开发，这会打乱程序在其他平台上的构建；更糟糕的是，如果当前没有在多平台上进行开发，当想要移植代码到其他平台的时候，也许会被一些麻烦所困扰。

除了会带来不必要的信息而外，应用预编译头文件的另一个缺陷就是，不同平台之间或者编译器的移植问题。没有预编译头文件的支持代码依然可以顺利地被编译，因为每个文件都包含了预编译头文件 precomp.h，这个头文件包含了所有需要用到的头文件。问题是 precomp.h 包含了整个程序用到的所有的大型头文件，并不仅仅是每一个单独的.cpp 文件需要的头文件。这就意味着，如果一个工程是立足于使用预编译头文件的，在不支持预编译头文件的环境中对它进行编译，其速度可能会比一开始就不使用预编译头文件还要慢的多。

为了缓和这些矛盾，在不支持预编译头文件的平台上，可以用所需包含头文件的最小集合来代替预编译头文件，就像下面这些代码：

```
// Precomp.h
#include "LargeAPI1.h"
#include "LargeAPI2.h"
#include "LargeAPI3.h"
#include "LargeAPI4.h"
```

```
#include "LargeAPI5.h"

// MyClassA.cpp
#ifdef PRECOMP_SUPPORT
#include "precomp.h"
#else
// This file only needs LargeAPI3.h
#include "LargeAPI3.h"
#endif

#include "MyClassA.h"
#include "MyClassC.h"
#include "SomethingElse.h"

// The rest of MyClassA implementation
// ...
```

15.3.6 Pimpl 模式

如果按照上面这些建议去做，则头文件是独立的（它们不依赖于为它们而包含的其他头文件），它应该只被包含一次，并且只包含能够保证文件正常编译的最少头文件。除此而外，如果编译器支持预编译头文件，那么应该尽可能高效的利用它。

即使做了上面这些调整，一个大的工程还是有可能编译的很慢，或者一个很小的改动都会导致很长时间的编译。需要为这些问题采取一些措施。没有人喜欢面对这样的工程，即使是一处小小的修改也要花费几分钟的时间或者每次重新编译都要花上一顿饭的时间。

引起这一问题的原因可能还是由于有太多混乱的头文件。这时候就需要看编译器是否具有按照预处理程序打开文件的顺序显示所有头文件的功能。如果每个.cpp 文件都包含了几十甚至数百个头文件，那么或许可以对程序的设计进行改进。反之，如果每个.cpp 文件只包含少数头文件，则问题出在其他地方。或许代码太多，而这本身就要花费很长时间。在这种情况下，对硬件进行升级，提高 CPU 或硬盘驱动器的速度或许比其他方法更有用。

试图改进物理结构之前，首先要看一看工程是否需要一个更好的逻辑结构。如果整个游戏只有一个工程，并且包含了几千个文件，它或许应该被细分为更小、更独立的系统。一些通常应该被分割的系统有渲染、输入、声音、AI、碰撞检测和物理模型。

将一个工程分成几个独立的子系统有很多好处（这一点会在本章后面的内容中详细了解）。一个最主要的优点就是减少了不同子系统中文件之间的物理联系。特别是当子系统使用了抽象接口或表面模式（façade pattern）来对程序其余的部分隐藏自己的内容时，惟一需要担心的就是同一子系统内不同文件之间的物理联系，而这是比较容易处理的。

假设问题仍然是包含文件太多，那么 Pimpl 或许会有所帮助。Pimpl 是 Private Implementation 的缩写，又叫 Cheshire Cat 模式，首先看一个用到 Pimpl 的例子。这个例子

在前面的 GameCamera.h 头文件的基础上做一点小小的修改：

```
// GameCamera.h
#include "GameEntity.h"
#include "CameraLens.h"
class GamePlayer;

class GameCamera : public GameEntity
{
    GamePlayer * GetPlayer();
private:
    GamePlayer * m_pPlayer;
    CameraLens m_lens;
};
```

这段代码与以前的 GameCamera.h 头文件的惟一不同就在于增加了一个额外的成员变量 m_lens，这个变量代表了摄像机所使用的镜头，它可以是一个完全透明的镜头，当可以使用彩色镜头为它添加颜色，也可以使用黑白镜头滤掉其他的颜色，甚至可以使用动态镜头在玩家被袭击的时候让红色闪过整个屏幕时。更好的还可以一次使用多个镜头，但这里要限制它的使用以便使例子简单一些。

将变量 m_lens 添加到头文件中的结果是，不得不包含声明 CameraLens 类的头文件。不能再像前面那样使用前向声明了，因为要处理的是一个实实在在的变量，而不是一个指针或引用。对于所有加入到类中的变量来说，情况都是一样的。

这是非常令人沮丧的，因为 m_lens 是一个私有变量。这就意味着只有 GameCamera 类可以访问它，所以让人感到失望的是不得不在 GameCamera.h 文件中包含 CameraLens 类的头文件，并让所有使用镜头对象的文件也都包含它。可以看到，如果 GameCamera 是一个在代码中很多地方都用到的常用类，那么包含文件的数量就会迅速地增加。

Pimpl 模式可以避免在头文件中包含那些只被私有变量用到的头文件。为了实现这一目标，将所有的私有代码放入一个简单的结构中，并让它与对象同时生成和销毁。这就是包含镜头的 Pimpl 查找 GameEntity 类的方法：

```
// GameCamera.h
#include "GameEntity.h"
class GamePlayer;

class GameCamera : public GameEntity
{
    GamePlayer * GetPlayer();
private:
    GamePlayer * m_pPlayer;
```

```
    class PIMPL;
PIMPL * pimpl;
};

// GameCamera.cpp
#include "GameCamera.h"
#include "GamePlayer.h"
#include "CameraLens.h"

struct GameCamera :: PIMPL
{
   CameraLens m_lens;
};

GameCamera :: GameCamera ()
{
    pimpl = new PIMPL;
}

GameCamera :: ~GameCamera ()
{
    delete pimpl;
}
// Rest of GameCamera implementation
```

注意，现在可以把对 CameraLens.h 的包含从头文件中删除了，这正是我们的目标。如果还有其他需要包含头文件的类，也可以把它们添加到 Pimpl 结构中。

使用 Pimpl 模式，通过增加额外的复杂性和对象私有代码的动态分配减少了类之间的物理联系。额外的复杂性并不算太严重。Pimpl 只需要创建一次，不管它包含多少变量，访问它们只需要将对象的名字作为前缀加到前面即可。

更加恼人的缺陷是由于对象使用 Pimpl 模式而导致的动态分配内存。这对那些实例个数很少的大型对象是非常好的，但对那些拥有成百上千个实例的小型对象却可能会有问题。如果在这种情况下还要使用 Pimpl 模式的话，也许应该用第 7 章中讲到的内存池系统。

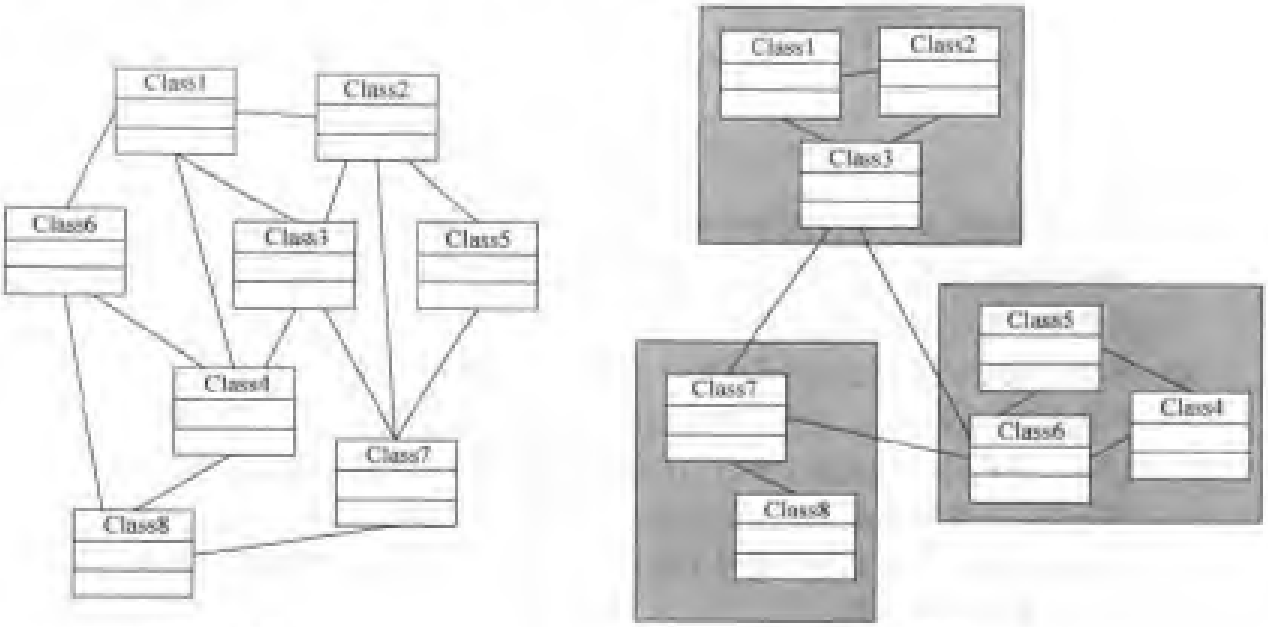
15.4 库

用于小型代码库的技术处理大型工程时经常会陷于困境。多数使用 API 或在网络上使用的源代码的规模都是很小的，它们被简单的放在一起，仅仅是为了证明一些特定的特征或是说明一个概念。根据它们的大小，这些小型工程被构造成一个单个的系统并被编译成一个可执行文件。

这些方法对那些相对较小的代码库可以工作的很好，但对于如今大多数游戏项目，它们的规模一般都很大，这样的方法就不太适用了。文件之间的联系将很快变得无法处理，编译时间会迅速增加，原本简单的问题会变得很复杂，程序的质量急剧地下降。

对于大型工程，将较大的代码库分成相对独立的子系统通常是较好的方法。每一个子系统是一组具有逻辑上相互关联的目标的源代码，它对外界暴露最少的函数，却允许每一部分使用它的全部功能。子系统被编译成独立于可执行程序的动态链接库或者静态链接库。这些库被链接到主程序中从而形成完整的程序。在程序开发过程中经常生成库的部分包括内存管理、图形渲染、输入控制、碰撞测试、物理响应、动画、音乐、声音重放、用户界面和一般的 AI 函数。

用这种方法将子系统划分成库的一个好处是减少了类和文件之间的联系，从而改进了工程的物理结构和逻辑结构。图 15.1 说明了将工程划分为库可以减少关联，从而为其他代码提供一个简单的接口。



（a）不使用库　　（b）使用库

图 15.1　类的关联

不将工程细分为库也可以达到相同的效果，但维护起来会相当困难。我们必须逐渐了解，强制进行人工分割的方法。因为在真正的工程中，会因为存在大量关联而变得混乱不堪。通过明确的创建子系统，可以强制进行划分并确保不会引入多余的关联。

现在从较高的层次上观察库，这样才能发现它们之间的关联关系，保证库之间没有循环依赖关系是非常有用的，虽然并不是绝对必要。这就意味着一个库可以与它下层的库建立关联，但却不能依赖于它上层的库，图 15.2 是一个正确的库的组织结构的例子。

用这种方式组织库可以防止代码库变得混乱不堪。通过使用库并包含所有与它相关的库，可以独立的处理其余代码中的任何部分。这种组织形式将会带来一系列好处。

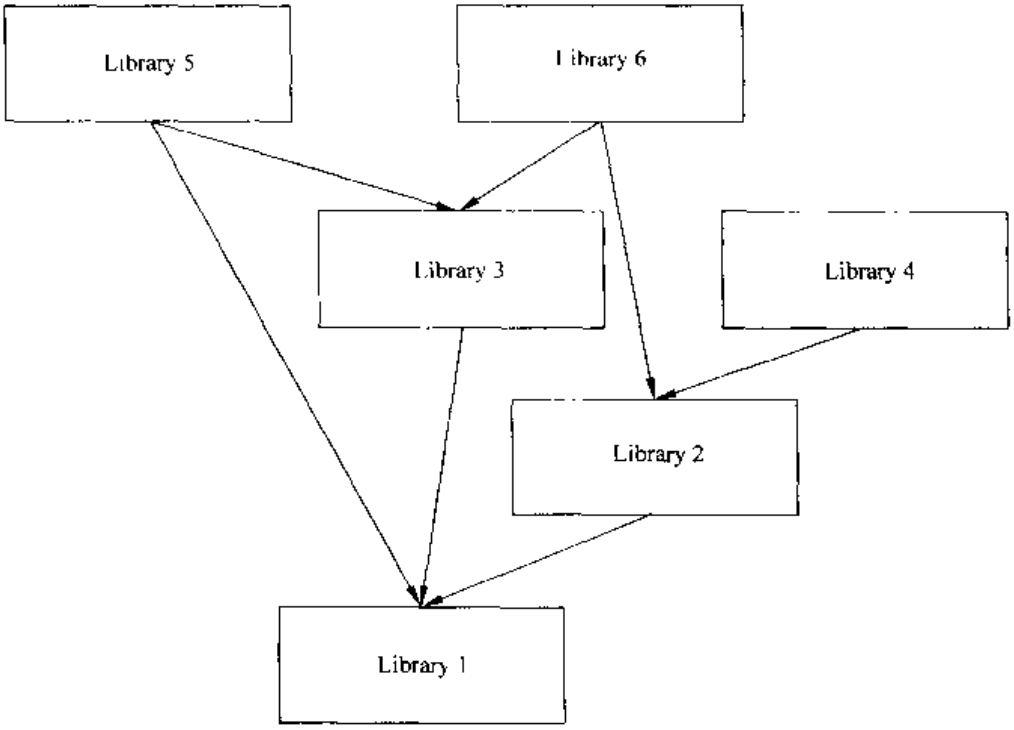


图 15.2 没有循环依赖关系的库的关联

- 可以采用自底向上的方式单独测试每一个库。这在试图定位一个 BUG，但又不知道它到底是在底层还是上层的时候显得特别有用。甚至可以创建一个依赖于某个库的测试程序来检查它的执行情况。
- 内部工具可以挑选它们所需的库。当处理大量代码的时候，这些工具或是包含所有的游戏代码（以后会带来代码膨胀，速度变慢等问题）或者不与游戏共享任何代码，这将妨碍清楚地了解它们在游戏中的作用，并造成代码的重复。利用分层库方法，一个需要图形渲染的工具，可以只包含该库以及在它之上的所有库。一个更复杂的工具可能包含更多的库，而一个几乎需要所有功能的工具则可能要包含大多数的库和代码。
- 很容易共享已编译好的代码。使用分层的库系统，可以编译一些低层次的库并将它们分配给其他部分使用。按照这种方式，程序员（只要他们做的是本地修改）只需要编译他们正在处理的库，而不是整个工程。这样就能节约大量的时间。进一步利用这种思想，可以让自动编译系统每天晚上重新编译所有的库，在第二天早上将编译好的新版本提供给程序员使用。
- 区分引擎和游戏专用代码变得更简单。当代码用相对独立的库进行构造时，游戏和引擎之间的界限成了一个天然的分界点。这样，只要稍加努力就可以将与游戏相关的库和游戏专用的库区分开，这将允许很容易地在不同的工程之间重用代码，甚至很容易地利用已有的代码库新建一个工程，因为很容易区分哪些代码是游戏专用的，并且需要被替换掉。
- 可以在代码中采用一些模块化的方法。如果代码本身就是结构化的，并且在单独

的库中，那么使用新的代码来更新整个库就变得相对简单了。这就允许像游戏本身一样定制引擎的许多部分。例如，几乎不需要更改任何代码就可以通过更换库将 portal visibility 系统变成 quadtree—based 系统。还可以使用新的目标平台库来代替某一个特定的库，从而使跨平台开发变得更加简单。

15.5　配　　置

在真正的工程环境下，只用一种方式来编译游戏是不够的。可能需要根据不同的需要选择不同的设置。每一个不同的设置就做一种配置。每种配置将使用不同级别的错误报告、代码优化和调试工具。

在配置中为所有的可执行文件和库加上后缀是一个不错的想法，这样做可以避免让它们使用相同的名字。利用这种方法可以在同一个目录下生成游戏的几个可执行文件，每次根据不同的情况选择所要运行的程序。

通常情况下每个工程至少要有两种配置：调试版本（Debug 版）和发行版本（Release 版）。

15.5.1　调试版本

调试版本通常只供程序设计人员在调试程序，或者向程序中添加新的特性时使用。这种配置完全不用编译优化功能，而且还包含所有的调试信息。这样就可以很容易地在编译器中进行单步运行，并能在编译器窗口中查看所有的变量。

这一配置有着各种各样的安全防卫和保护，以便在代码中出现逻辑漏洞的时候能够有所察觉。但是要付出性能上的代价。例如，这一配置可以自动对每个数组调用进行越界检查，或者将闲置的内存空间用某种位的组合格式填满，以便更容易地发现哪些指针指向的空间已经被释放。

这种配置下的性能是比较差的，很可能会比最终的产品慢两到四倍。这将会使运行的效率变慢，以至于由于系统速度的原因可能会导致游戏进行的不那么流畅。但在这种配置下性能本身并不是关键。此外，调试配置还需要更多的内存，因为它有许多额外的代码和用于调试的符号。

15.5.2　发行版本

它是与调试版本相对的另一种配置，它实现了所有的优化功能，去掉了调试辅助功能及其他与调试有关的方面等。在这种配置下生成的可执行文件通常要送到制造商那里去生产最终产品的 CD-ROM、DVD 或磁带，这就是用户最终所要得到的。

在这种配置下可以得到尽可能好的性能，但却几乎不可能对它进行调试。如果所有的

调试代码被删除并且游戏崩溃了，它也不会报告任何有助于捕捉 BUG 的信息。

15.5.3 调试优化版本

调试版本和发行版本是相对的，它们都很实用，但还需要一些折衷的解决方案。这个中间的解决方案在性能优化和调试方便两个方面获得了平衡，并且大多数游戏开发工作都可能用它来进行。

这个配置通常包含代码调试信息，以及所有的调试工具，像游戏内部的控制台或是完整的出错报告等，但它同时也启动了优化功能，这些功能可以让程序获得尽可能快的运行速度。这种配置将比发布版本占用更多的内存，因为它要用到所有的调试符号。

有些工程使用另一种中间配置，这种配置没有任何编译信息，但却能在游戏崩溃的时候生成有用的出错报告。这一配置可以用于排除发行版中出现的问题。有时候游戏的发行版中会带有一定等级的出错报告，这主要取决于游戏和具体平台。在这种情况下，出错报告会在发行版本下建立，因此就不再需要中间版本了。

15.6 结 论

在本章学习了源代码的物理结构和逻辑结构之间的区别，以及一个好的逻辑结构却得到一个糟糕的物理结构的几种可能情况。物理结构是指文件之间而不是类之间的联系。一个好的物理结构可以获得较快的编译速度，缩短修改程序所用的重新编译时间，并且简化了代码的测试和维护。物理结构在大型系统中的作用比在小型演示程序中要重要的多。

接下来讨论了将一个类分成多个文件时推荐使用的组织形式：一个文件用于类的声明，而另一个用于类的实现。对包含头文件的管理是生成好的物理结构的关键，所以学习了使包含免于陷入混乱状态的几个推荐的规则，包括如何减少包含头文件的数量以及如何避免头文件与其他头文件之间建立关联。还讲到了两种缩短编译时间的技术：预编译头文件和 Pimpl 模式。

一个有着好的、先进的组织结构的工程，通常分成相对独立的几个库，并且具有清晰的关系链。这样可以有许多好处，比如代码重用，库的替换，易于测试以及生成工具或者运行测试程序时，具有使用库的一部分能力等。

最后，任何大小合适的工程都需要有多种配置来生成不同的可执行文件或库。最常用的是调试版、优化调试版和发行版。

15.7 阅 读 建 议

主要的 C++文献似乎只涉及程序的逻辑结构，也有一些书在一定程度上讲到了物理结

构，John Lakos 的书是论述得特别详细的一本。

Lakos , John , Large Scale C++ Software Design , Addison-Wesley,1996.

Stroustrup, Bjarne, The C++ Programming Language, 3rd ed. ,Addison-Wesley, 1997.

下面是一本非常有名的书，它讲到了许多实用的模式。它对表面模式（Façade Pattern）的讨论刚好和本章的内容相吻合，这种模式用于减少库或模块中文件关联的数量。

Gamma, Erich, et al., Design Patterns,Addison-Wesley,1995.

有许多书和文章谈到了 Herb Sutter 的 Pimpl 模式。下面这本书的几章中讨论了这一主题。

Sutter, Herb, Exceptional C++,Addison-Wesley,2000.

第 16 章　防止游戏崩溃

本章将讲述：

- 使用断言
- 刷新机器状态
- 处理“坏”数据

可能现在已经意识到这一点，C++并不是很“宽容”的（这里是指容错性）。必须对所要做的事特别地明确。有时候，与好的软件工程的规则相反，甚至必须写一些多余代码，比如要将函数名和参数同时写到函数的声明和定义中去。

C++在运行时也同样是不“宽容”的。一个小小的计算上的错误都会导致程序的崩溃，好的时候会给出一些错误信息；糟糕的时候甚至会造成死机（玩家很可能卸载掉游戏，并要求商家退货）。

显然，必须尽量避免后一种情况的发生。本章将介绍几项防止游戏崩溃的技术。将了解如何有效利用断言函数来检查 BUG 的产生，如何保持机器处于刷新的状态来避免运行以后出现的 BUG 以及如何在不牺牲性能和安全性的条件下处理函数中的坏数据。

16.1　使 用 断 言

断言语句是标准 C++类库的一部分，很快将成为最常用的工具之一。简单地说，它所实现的就是在条件不满足时立刻终止程序的运行。

16.1.1　什么时候使用断言

要防止游戏崩溃，为什么需要使用断言呢？在某种程度上，是在通过自己结束程序来避免游戏崩溃。主要的不同在于前者是在自己的控制下，可以收集一些对调试有用的信息，这些信息应尽量在发生错误的地方获取。捕捉到一些很早以前就产生，但却完全没有被发现的错误是最令人头疼的了。通过使用断言函数，在刚刚出错时就收到信息，这将使调试工作变得更简单。

下面是程序中经常使用断言函数的地方：

- 如果有无法处理（或是不想处理）的参数传到某个函数中时，则终止程序的运行。每个函数都必须处理一些参数，尤其是类的私有函数或是最终用户不可见的函数。

及时察觉函数的误用要比垃圾产生后再进行清除好的多。

这些程序节点经常称为函数预处理，每一个函数都有一系列先决条件。如果这些条件不满足，函数将不能执行。

例如，一个负责向玩家的财产清单中添加对象的函数使用一个指向游戏对象的指针。为了简化程序，从一开始就认定空指针是非法的，因为它没有任何意义，并且也不允许那样做。在函数的开头使用断言函数可以确保指针是非空的。

```
void Inventory ::AddItem ( Item * pItem) {
   assert ( pItem != NULL );
// Add it to the inventory here, since now we know for sure it is not NULL
// …
}
```

- 检查程序状态的一致性。在进行一项操作之前，有时需要确认一下它能否顺利地执行。通过使用断言函数我们可以在问题出现以前就捕捉到它，这对于以后的安全维护也是很有用的：现在我们很清楚我们所要做的事情，这种检查也许还不是很必要，但将来仍然有机会修改程序。在这种情况下，这些额外的断言函数可能会对早日发现问题发挥着重要的作用。

例如用于销毁实体的函数。根据以往的经验，有时候会在不知不觉中收到一些终止运行的消息，但无法预料它们会在什么时候出现。所以应该对这种可能性有所防范。如果这种情况发生的话，我们应该及时对它进行处理来避免 BUG 的出现，而不是将它忽略或触发异常。

```
void GameEntity :: HandleKillMessage ( const Msg & msg ) {
   assert ( !IsDead() );
   SetIsDead (ture);
   //Do the rest of the killing process here…
}
```

- 检查一个复杂的算法是否正按照所希望的方式运行。这通常被称为“健壮性检查”。我们知道，当执行完一系列操作之后，某些东西应该处在一定的状态之下。但是由于程序特别复杂，所以很难保证它的正确性。为了确认一下，可以插入一个断言语句，这样就能立即知道错误是否发生。

这些是后处理，与预处理相对应，它们同预处理一样都要检查是否满足特定的条件，但它们发生在函数的最后而不是开始。这一点与预处理不同，对它们的检查并不像检查前提条件那样重要，但它可以让我们更好地捕捉开发过程中的 BUG，并让我们对代码的执行更有把握一些。

下面这个函数用于处理队列中所有的消息。但是当一个消息被处理的时候，可能有更

多的消息被发送。该函数的后处理用于确认当函数结束后，队列中没有其他的消息。可以用断言函数来实现这个后处理。

```
void MsgQueue :: ProcessAll () {
    // Do all the processing here. Maybe frome different queues,
    // so it is not a straightforward process
    // ...

    // Before we finish, double check that there are no message left.
    assert ( IsEmpty() );
}
```

- 如果一个函数执行失败并且无法修复，则终止程序的运行。

有时候会发生一些非常糟糕的事情，试着分配一块内存却得到一个空的指针，或者试着移动一个敌人实体却发现它已经不存在了。显然这是代码中出现了严重的问题。这时候就应该借助于断言函数了，它可以帮助确定问题的所在。

有时候希望程序自己能够检测到出错的部分，并自动地进行恢复，这是一个美好的愿望。但对于游戏来说，常常是不必要的。尽管如此，也并不意味着不会去恢复它，下一部分将给出一些建议来说明什么情况下不使用断言。

16.1.2 什么时候不使用断言

到现在为止，好像所有不能确定的地方都要使用断言。一般来说的确是这样，但有些情况下需要更优雅地处理某些错误。

从以往的经验总结出来的一个好的方法，设计者只有在遇到程序 BUG 的时候才触发一个断言函数（或者是彻底结束程序）。换句话说，使用游戏或工具的人不会导致程序的崩溃。如果遵守这一规则，将会给每个人都带来方便，并且会尽可能减少由游戏或者工具工作不正常所带来的停工期。这意味着什么？这意味着在下面这些情况下都不应该使用断言函数：

- 打开或载入一个并不存在的文件。
- 试图载入一个格式错误或以前版本的文件。这并不是说应该始终支持以前版本的文件。只是不应该因此而终止游戏。
- 输入错误数据。例如，如果用户不能区分远程飞机和近程飞机的镜头设置，它们给远程飞机设置的数字比近程飞机还要小，这时最好的方法是从一开始就在用户界面上防止这种输入，而不是使用断言函数。
- 游戏对象没有被正确的载入。如果它不是游戏运行所必需的对象，只需要弹出一个警告对话框并让游戏继续运行。如果它是必须的（比如玩家对应的镜头），那

么应该输出一些消息来说明问题的所在并终止游戏。设计人员遇到它的时候要比遇到一个断言函数要好的多。

另一方面，如果一个测试者试图让人物进入一个并不存在的房间，这时候最好使用断言函数，因为这应该是不可能做到的事情。简单地说，断言函数就是程序员为自己设置的信息。如果一个断言函数被触发，那么应该立刻提交给程序员去修复它。

16.1.3　自定义断言函数

断言是标准 C++库的一部分，这对我们来说是一件好事。我们可以及时方便地使用这个函数。然而，如果要在大项目扩展它的一些应用的话，它就显得有点过于简单了。我们需要获得更多的功能，这时就要生成自己的断言函数了。

我们所需要的是更多的信息和更好的灵活性。记住断言函数是发给程序员的信息，如果是在调试器里运行程序，我们所要做的只是使用断言函数提供的默认功能。但是如果它发生在测试部门测试游戏的时候，那么就需要用于确定问题的所有信息。即使是在编译器中运行程序，程序员也不可能立即观察到致命的问题。所以显示相关的信息就变得更加重要了。

下面是用户自定义断言函数时需要考虑的一些可能的选择。

- 收集更多的信息并将其显示在屏幕上。默认的断言函数会终止程序的运行，显示错误的状态，有可能发生错误的文件以及错误的所在的行号。这对基本的应用来说是不错的。但是还可能需要更多的功能，比如查看堆栈、注册表状态以及所生成的版本。
- 将信息保存到文件中。不仅需要显示错误信息，而且还需要将信息保存到文件中，这样就可以通过 E-mail 将其发送给其他人，或者保存到 BUG 中以便日后进行修补。在文件中，需要保存显示屏上的所有信息，或者还有一些用于排查的代码和可能有用的内存转储（比如堆栈、指令指针等）。

此外还可以做的更多：对运行平台作一个主存储器信息转储（core dump）或类似的一些功能。主存储器转储是将计算机的状态以二进制的形式进行一个快照，以便以后在不同的机器上恢复相同的状态来达到调试的目的。

- 直接用 E-mail 发送信息。一个好的功能是允许所有的断言函数信息立即通过 E-mail 进行发送。这通常只对 PC 机是可行的，而对游戏终端却不行，但是它却是非常方便而又值得考虑的。另一方面，如果使用 BUG 库的话，可以将它们和测试人员的评价以及他们对导致断言函数的情形的描述一起存入 BUG 库中。

在考虑用户自定义断言函数的时候，经常会遇到的一个选择是：当一个断言函数被触发的时候，是否会让程序继续运行。考虑这一选择的理由是，有时候虽然触发了断言函数，但游戏还可以继续运行，所以为什么要强制别人去终止程序呢？这是一个典型的错误想法，主要理由如下。

首先，断言函数是用来检查那些允许游戏继续执行的必须正确的条件的。代码的正确执行都取决于这些条件，所以继续执行几乎必然会导致错误的发生。由于非终止断言函数的存在，开发者不再依赖于断言发生时程序的情况了，迫使他们检查代码，这将使代码变得更加混乱、臃肿、缓慢和难于维护。

非终止断言函数的另一个结果是：它们将不可避免地成为标识是否出现了错误和警告消息。例如，如果一个文件不存在或者一个对象没有正确地初始化，出于压力之下的程序员没有使用“发现错误并报告”这种模式，而是转向使用一个非终止的断言函数，允许用户单击“继续”按钮运行下去，这将导致一系列的问题。错误信息不是分批处理的，所以在玩游戏之前可能会有大量的验证框，错误可能没有被正确的报告或保存，真正有用的断言可能就会被湮没在大量混乱的没有意义的断言中了。

最后也是最严重的问题。假如一个程序在非终止断言执行以后的某个时候崩溃，要判断它是到底由于新的代码中的 BUG 造成的，还是由于断言函数的继续执行引起的，是非常困难的。

16.1.4 对最终发行版的处理方法

一般来说，所有的断言函数调用都会在最后的版本中删除。原因是所有的 BUG 都已经被剔除了，保留断言函数只会影响游戏的运行效率，并且会使内存消耗变大（由于所有的断言函数语句及其相关信息）。换句话说让游戏非正常地工作下去或者崩溃比向玩家显示一个错误声明要好的多。

然而，事情并不是那么简单。我们的决定将取决于我们的目标平台和开发计划。

一个严重的问题是无论对游戏做多少测试，总会有一些人遇到触发断言函数的情况。如果对此不大相信的话，试想一下可能的情形：假设有一个非常大的测试部门，有 40 人对游戏进行最后的测试，测试要持续 20 个星期，每星期 40 小时（忽略这 20 个星期里游戏的改动情况，认为它是理想状况）。总共是 32 000 小时的测试，约相当于三年半的时间。

当排除所有的漏洞以后，游戏被发布并立即投入市场，第一个月卖出了 100 万套。现在让这 100 万人平均每天玩 1 小时，两个星期下来，就相当于 1400 万小时或者 1639 年的测试。比详细测试阶段所花的时间高三个数量级。而这才只是第一个月。这个数字是保守的，真实的数字可能要高的多，这取决于销售情况和人们使用的情况。这还没有考虑到不同用户的配置，不同性质的硬件以及有缺陷的存储介质。

做开放的 β 测试时，情况会有所好转，但这种测试并不是在所有的情况下都是可行的（比如，游戏终端上的游戏）。即使在 β 测试中，经过测试的也只不过是游戏的一小部分，因此，BUG 还是会在没有引起注意的其他部分出现。

换句话说，发行一个没有 BUG 的游戏是不可能的。只能去接受它、克服它、尽最大的努力去解决它。

需要考虑一下一旦游戏被发布以后，如何对 BUG 进行处理。我们承认发行的游戏中存

在 BUG，并不代表我们可以不去修复它们或者发行一款明知道有 BUG 的游戏。我们仍需尽我们最大的努力来修复 BUG，但也不得不承认并没有发现所有的 BUG。

正如前面所提到的，通常处理 BUG 的方法就是将它忽略。所有的断言函数调用都被删除，我们做着最好的打算。如果发生什么问题，有可能会产生一些怪异的结果，也有可能游戏会冻结掉，这对玩家来说是扫兴的事情。

作为另外一种选择，特别是在经常为游戏打补丁的 PC 机平台上，可以通过单击按钮将断言函数与用 E-mail 发送崩溃信息（包括用户配置和驱动版本信息）的功能一起保留下来。游戏发行几个星期以后，可以处理所有的错误报告并排除大多数问题，这只是特别针对 PC 机游戏的一种选择，因为有太多不同的用户配置形式以至于不可能在发行之前就对它们完全的哪怕是对典型性的配置进行测试。

游戏终端配置更受限制，所以硬件测试要彻底的多。此外，在游戏终端游戏中，打补丁是不能接受的，尽管也有可能实现，因此打补丁不是一个可行的选择。

如果既不想将断言报告给玩家，同时又不想崩溃，应该怎样做呢？对此没有确定的答案，可以考虑下面这些的可能方法。

- 从发生错误的地方恢复。如果在图像子系统中获得一个错误断言。可以试着将它完全关闭，然后重新启动，并继续运行游戏。实现这一点是非常困难的，因为一旦获得了错误断言，系统的状态很可能已经被破坏了。
- 重新启动游戏终端。可以试着保存游戏（假设断言函数没有影响游戏保存数据的完整性），重新启动游戏终端，尽可能快的重新载入它。玩家可能会被中断几分钟，但这比游戏崩溃要好的多。
- 结束关卡。如果发生问题，可以结束本关卡，重新返回上一级。甚至可以早点结束该关卡来补偿游戏内部出现的问题。

一个折衷的解决方案就是，在最后的发布版中完全删除断言函数，但处理所有的内部异常。通过删除所有的断言函数我们做了最好的打算。也许一些情况会导致断言函数被触发，但那并不会使游戏终止。然而如果有非常严重的情况发生，这主要取决于你的平台，则很可能会导致异常的产生（例如访问错误的内存区域、错误的使用硬件等），我们可以捕获这个异常并尽最大的努力对它进行处理。

16.1.5　自定义断言函数的代码实例

一旦游戏被发布，最终采用什么方法来处理错误呢？显然用户自定义的断言函数是非常有用的，可以根据构建的类型改变断言函数的实现代码，完全终止它的使用以避免付出性能上的代价。

这些实例代码只在调试版时起作用，在发行版时它将被完全关闭。可以通过预定义 DO_ASSERT 独立灵活的控制它对于某种配置的开启和关闭。

```
#ifdef _DEBUG
#define DO_ASSERT
#endif

#ifdef DO_ASSERT
#define Assert ( exp,text ) \
    if ( !(exp) && MyAssert (exp, text, _FILE_, _LINE_)) \
        _asm int 3;
#else
#define Assert ( exp,text );
#endif
```

这个宏工作的方式是首先判断表达式，如果表达式的值非真，将以这个表达式、描述文本、发生错误断言的文件和所在的行为参数来调用 MyAssert 函数。如果函数要终止程序的运行并返回到调试器中，它将返回 true，否则将返回 false 并且允许程序继续运行，那就是 if 语句后面_asm int 3; 行的目的。在 PC 机中，这将导致程序跳到调试器中，如果调试器没启动，调用将被忽略，程序会继续运行，否则系统会自动调用类似的函数，允许你跳出执行。通过在宏语句中加入_asm int 3; 行，调试器将在调用断言函数的地方跳出来，而不像标准断言函数那样在内部实现文件中跳出。

MyAssert 函数所实现的功能完全取决于我们自己。它可以简单的在调试面板中输出一些说明性的文本以及文件名和行号等，或者也可以通过对话框来输出这些信息，当然还可以将它保存在文件中以备日后处理。

16.2 刷新机器状态

经常会有这样的情况：游戏运行了一段时间没有出现任何问题，然后开始测试，持续运行了五六个小时或者更长的时间以后，报告出现致命错误。出了什么问题呢？

这可能是一些最具破坏性的并且很难修复的漏洞，因为这些漏洞经常反复出现，它们只有在长时间持续运行之后才可能出现，如果幸运的话可能是 1 小时，否则可能会是几天。这些错误可能由很多原因引起。

16.2.1 内存泄漏

内存泄漏在 C++中是很常见的。一些内存是动态分配的，却没有被正常地释放。大的错误通常可以在开发过程中很快的被找到，但有一些小的内存泄漏有可能偷偷溜过。它们在短时间内不会有什么致命的影响，但一旦它们开始积累，就会影响到系统的运行，最终系统会因内存不够而崩溃，或者与虚拟内存进行交换，从而影响系统性能。

假设每秒钟泄漏 200 字节内存（可能是由于 AI 更新它的寻路信息而造成的）。5 个小时以后，泄漏的内存就会达到 3.4MB，这对于内存较大的 PC 机也许不算很大，但对于资源有限的游戏终端（游戏机）来说却是一个严重的问题。如果内存不是每秒泄漏一次而是每帧画面泄漏一次的话，几小时以后即使是很少的泄漏，对于 PC 机来说也是一个不小的问题。

16.2.2　内存碎片

动态内存是从堆中分配的。首先，堆中的内存作为一个连续的整体都是可用的。由于分配的需要以及释放内存，堆被分成许多不连续的小块，即出现很多碎片。这就意味着空闲的内存块将分布在许多已经被使用的内存块之间。最后，这将会产生与内存泄漏类似的后果，因为有可能需要一块很大的内存，但却找不到一块这样大的连续内存，因而导致了内存分配的失败。

这个问题在游戏终端上比在 PC 机中更加严重。因为 PC 机的内存一般较大，有时候开发人员可以不使用动态分配内存来避免这样的问题（包括内存泄漏），但这是一个极端的方法。它会降低资源的利用率，并且还有很多限制，通常更倾向于内存池等其他的解决方案（详见第 7 章内存分配）。

16.2.3　时间的表示

有时候持续的游戏时间在游戏中用浮点数表示。原则上，这听起来是一个不错的想法；直到仔细研究它才会发现许多不妥的地方。因为浮点数的本质，数字越大精度就越低。从秒开始，将会有 32 位用于秒以下的存储。两小时以后秒以下的精度降低到 11 位，而运行 18 小时以后精度会降低到 8 位，这就意味着最小的时间增量大约是 8 毫秒。这时，情况就变得比较危险了。时间的变化不再可靠，在游戏中会产生一些奇怪的结果，特别是对于那些对时间比较敏感的系统，比如物理和冲突检测等。

另一个常用的时间表示方法是用 32 位的整数表示从游戏开始所经过的毫秒数。整型不会丢失精度，但它在 49 天以后就会超出所能保存的位数。这通常对于大多数游戏都是足够的（那些需要一直运行的游戏服务器除外）。

16.2.4　错误积累

另一类由于长时间运行游戏所导致的错误类型是不精确的数据运算积累而来的。有时候不是从粗算的结果中重新计算一个值，而是在它的基础上增加一个新值，这种情况可能发生在镜头的位置和方向、游戏对象的位置或其他一些方面。有时候计算上的错误会不断积累，运行几个小时以后就会出现一些奇怪的结果。

16.2.5 采取哪些措施

我们已经看到，经过长时间运行以后，除了可能发现一些常规 BUG 以外，游戏本身也会出现一系列问题。有什么方法可以避免这类问题吗？

答案就是让机器保持刷新状态，尽可能让机器保持接近游戏开始时的状态。在游戏终端游戏中，实现这一目标的最好方式就是在跳关（进入下一个关卡）的时候重新启动机器。通常，游戏终端都具备快速启动模式，用户甚至不会意识到机器的重新启动。这样做是解决所有这些问题的最好方案，因为从技术上来讲，游戏只会持续运行一个关卡。所有的一切，包括存储系统和时间计算全部都会被重新设定了。

不幸的是并不是所有的游戏都可以在跳关的时候重启，在 PC 机上就不能那样做，而且在一些运行环境比较特殊（例如需要保持网络连接）的游戏终端游戏中也不能那样做。在这种情况下，必须人工实现一些功能。

在跳关或是其他方便的时候（比如游戏暂停或是系统没有完全载入时），应该尽可能多的处理某些系统的设置，至少内存和时钟系统应该是这样。内存堆应该被压缩，甚至还要进行某些形式的垃圾收集，或者对内存进行遍历以便检查内存泄漏，时钟也应该设置为特定的步长。或许还要对图像系统、声音系统、物理系统或 AI 进行重新设置来避免运算错误。这虽然不如重启机器的效果理想，但它有助于防止或者至少是延缓上述问题。

如果无法选择重启，同样可以选择一次性销毁关卡的所有数据。在某个关卡结束时，不用释放每一个单独动态分配的对象，相反只要重置关卡数据所占用的内存。只要没有指针或代码依赖于该区域的任何东西，一次性擦去这块内存的内容可以防止内存碎片问题，并且可以获得更高的效率。如果使用内存池（见第 7 章）来保存大量的关卡数据，做到这一点还是相当简单的。

16.3 处理“坏”数据

什么是“坏”数据，它有什么问题？让我们来看下面这个实现向量标准化的简单函数：

```
void Vertor3d :: Normalise () {
    float fLength = sqrt (x*x + y*y + z*z);
    x /= fLength;
    y /= fLength;
    z /= fLength;
    }
```

你的程序有可能会发出警告，我们有可能被 0 除，这的确是有可能的：如果有人对一个长度为 0 的向量调用 Normalize 函数，代码有可能将 0 作为除数，这就是“坏”数据。因

为从函数的角度来说这是没有意义的，标准化一个向量就是在不改变向量方向的前提下，将向量的长度设为一个单位。那么对一个长度为 0 的向量进行标准化又意味着什么呢？它本来就没有初始方向，因此毫无意义。

坏数据处理困难的一方面原因是因为它有传播的趋势。如果用一个类似上面的函数对坏数据进行处理，或者触发一个特定类型的异常（如果系统没有该类异常的触发器）或者计算出来的值为 NAN（Not A Number，非数据）。NAN 是一个用来指出无效结果的浮点型数据。问题是对 NAN 进行的任何操作都会得到一个 NAN，所以这些结果会随着所有的计算进行传播。它最初是在 IEEE745 标准中被设计成这种方式的，但它是为了对科学计算进行浮点标准化，而不是为了游戏开发。那么怎样解决这个问题呢？有两种对立的观点。

16.3.1　对坏数据使用断言

利用本章第一部分所讲的知识，通过断言避免坏数据出现在任何函数中。这种方法的优点在于，可以在坏数据出现的时候，做上标记并立刻对它进行修复。它的缺点与断言的一般应用相同，如果在游戏的最终版本中保留它们，就可能会在某些情况下触发它们；如果删除断言，函数会返回 NAN，并在游戏的其余部分产生不可预知的结果。下面是标准化函数的具体实现：

```
void Vector3d :: Normalize () {
    float fLength = sqrt (x*x + y*y + z*z);
    assert (fLength != 0);
    x /= fLength;
    y /= fLength;
    z /= fLength;
    }
```

让数学函数更容易出现问题的原因在于问题本身的性质。与大多数其他函数不同，它没有好数据和坏数据的范围，它们之间的区别是非常模糊的。例如，上面的标准化函数可以处理除了零向量以外的任何向量。类似的还会有不能处理两根平行向量或者两个处于空间同一位置的点的数学函数。比如一个游戏可能一直都在正常运行，直到玩家把镜头与其他实体排成一行，从而导致镜头向量与实体位置向量严格平行才会导致函数执行失败。而重新再产生类似的情况是非常困难的，这种非常少见的情形即使在内部测试时也是很容易被忽视的。

16.3.2　坏数据的复制

相反的观点则主张应该确保游戏中的函数不会崩溃，不会触发断言函数，但可能返回

毫无意义的值。那么应该如何标准化一个零向量呢？可以检测这种情况的发生，而不对向量做任何处理。不幸的是返回的向量并不是单位长度。另一种可选的方法返回一个沿着任意轴的单位向量，尽管向量的方向是毫无意义的，但它可以满足函数后处理的要求。下面是标准化函数复制坏数据的实现代码。

```
void Vector3d :: Normalize () {
    float fLength = sqrt (x*x + y*y + z*z);
    if( fLength > 0) {
    x /= fLength;
    y /= fLength;
    z /= fLength;
    }
    // Return a unit vector along the x axis if
    // this is a zero-length vector
    else {
       x = 1.0f; y = 0.0f; z = 0.0f;
    }
}
```

这种方法的优点就在于它可以让程序更加健壮。它不可能忽略任何返回 NAN 的数据（而不是首先忽略 NAN 参数）。不需要使用断言函数，因此即使有什么异常的情况发生，函数也能够对它进行适当的处理而不需要终止程序。

但是这种方法也不是没有缺点。这些函数都是很基本的，它们可以在每帧画面中执行很多次（很容易达到几千次）。向函数里添加额外的逻辑会使函数的执行变慢，甚至还会使它变大，以至于不能对它进行内联，从而显著的影响程序的性能。

另一个缺点就是它的返回值。虽然它不是 NAN 型，但至少它没有什么确定的意义，这将会在游戏中引发一些奇怪的反应。在这种情况下，假设要标准化玩家与一个实体之间的法线。由于某种原因，法线的长度为零，仍然要继续对它进行标准化。这个函数会返回一个沿 X 方向的向量，这显然是不对的，有可能会使玩家被墙卡住。

最后可能也是最大的缺点就在于它对固定结果的依赖。一旦所有使用坏数据的函数都可用，人们可能会不加思考的就使用它，用它们实现游戏代码的一部分。我们可能会对零向量进行标准化，可能会对负数计算平方根，用零作除数等。如果在某一时刻某些行为被更改（比如让游戏转到其他平台或优化函数），修改与之相关的代码将是相当困难的。

有意思的是，这是专门为游戏终端开发的一些标准数学库中的方法。他们意识到没有必要进行异常处理或生成 NAN，因为是在开发游戏。相反它们用可能的最好方法来处理数据。例如，在一些游戏终端游戏中，用零去除一个数字结果为零。当一个为游戏终端开发的游戏被移植到一个不同的平台上时就会产生问题，因为这将导致不断的产生冲突，直到逻辑错误被更正为止。

16.3.3　一个折衷的方案

到现在为止，所看到的两种选择对游戏开发来说似乎都不是很理想。它们都有一些很大的缺点。相对的给出第三种方案，一个好的选择就是综合上面介绍的两种方法。

为了发挥两种方法的优势，一个可行的方法是在两种方法之间切换。默认情况下使用第二种方法，并用最好的方式来处理坏数据，这将确保游戏不会被终止。但它会导致代码中出现没有意义的返回值。关键是要时常换到第一种方法，这样只要坏数据一出现就对其进行断言。每隔几周就更换一次可以让捕获绝大多数的坏数据。为了便于更换，最好是将特定的断言函数放在#ifdef 语句中，只在需要的时候才编译它们。

```
void Vertor3d :: Normalize () {
    float fLength = sqrt (x*x + y*y + z*z);
    #ifdef ASSERTBADDATA
    assert ( fLength > 0 );
    #endif
    if( fLength > 0) {
        x /= fLength;
        y /= fLength;
        z /= fLength;
    }
    // Return a unit vector along the x-axis if
    // this is a zero-length vector
    else {
        x = 1.0f; y = 0.0f; z = 0.0f;
    }
}
```

这种方法的优势在于它不鼓励向数学函数中传递坏数据（因为会依赖于没有意义的结果）；同时它保证一旦游戏被发售，即使出现某些情况程序也能正常运行。

主要的缺点还是由于额外的条件和不能内联所带来的性能损失。所以有必要为同一个操作提供两个函数：一个是安全的，因为做了很多的检查；另一个是速度较快但没有做检查的。问题是如何决定到底使用哪个函数。这需要一定的规则。只在那些已经确保没有坏数据的函数中调用速度较快的版本。这是非常重要的，否则，就得从头再来处理那些对坏数据调用没有安全检查的函数的代码。一个潜在的解决方案是只在数学库中使用不安全版本，而不让这些函数在游戏的其余部分出现，这将有助于纠正问题，但却妨碍了某些部分对快速函数的使用。生成的标准化函数的形式如下：

```
void Vertor3d :: Normalize () {
    float fLength = sqrt (x*x + y*y + z*z);
    #ifdef ASSERTBADDATA
    assert ( fLength > 0 );
    #endif
    if( fLength > 0) {
        x /= fLength;
        y /= fLength;
        z /= fLength;
    }
    // Return a unit vector along the x-axis if
    // this is a zero-length vector
    else {
        x = 1.0f; y = 0.0f; z = 0.0f;
    }
}

inline void Vertor3d :: NormalizeUnsafe () {
    float fLength = sqrt (x*x+y*y+z*z);
    assert ( fLength != 0 );
    x /= fLength;
    y /= fLength;
    z /= fLength;
}
```

16.4 结　论

本章的中心是关于如何避免游戏在运行时崩溃的技术。一旦发生崩溃，就会影响用户的体验（也影响我们的声誉）。第一部分介绍了如何在程序中高效地利用断言函数。了解了什么时候应该使用断言函数，以及什么时候应该使用其他方法。讨论了一旦游戏被发布，对断言函数的不同处理方法的优点，并介绍了一个在游戏中可以使用的非常基本的用户自定义断言函数。

第二部分解决了游戏运行一段时间后逐渐出现的一些问题。内存泄漏、内存碎片、时间误差的累计和在这些情况下所出现的一些问题。

最后给出了一些处理坏数据问题的解决方案。坏数据是指传递到函数中的没有意义的数据。一方面不想程序浪费时间，因此每个底层函数都能处理任何参数。另一方面希望确保游戏在某些情况下不会被终止。一个折衷的解决方案是每隔几个星期就转换到断言方式，来报告每个坏数据的实例，然后再换回忽略状态。通过这种方法有望使代码不断适应坏数据，但是仍然需要尽可能的在第一时间捕获到由于坏数据所产生的 BUG。

16.5　阅 读 建 议

下面是少数几本讨论如何使用断言函数调用以及为什么要使用它的书之一，似乎在其他的书中都将断言函数作为实现细节而将其忽略。

McConnell , Steve ,Code Complete ,Microsoft Press,1993.

下面这篇文章详细讨论了如何建立自己的断言函数。

Rabin, Steve, "Squeezing More Out of Assert ",Game Programming Gems ,Charles River Media,2000.

关于附带光盘

本书讲述了一些复杂的概念，本书所带光盘中包含了演示这些复杂概念的程序的源代码。所有的代码都尽可能地简单，以便让读者能够抓住问题的关键；但与此同时，它们又是一些非常实用的程序，可以反映不同方法之间的联系以及它们是怎样整合在一起的。

本书并没有将一页页大量繁琐的代码都放进来，各章节的内容中只包含一部分比较重要的代码段。其余的细节可以参考光盘中的源代码。如果喜欢读大量的源代码，可以打开源代码文件，一边学习本书，一边领悟代码。

使用源文件的最好方法是编译文件来验证它的运行结果，如果想检查某种操作的执行过程可以使用单步调试方式，另外也可以对它们进行修改来看看如何扩展和改进它们。

每个程序的源文件和工程文件都放在一个单独的文件夹中。

- 第 7 章：内存管理。一个完整的内存管理器的完整实现和一个小型的测试程序。
- 第 11 章：插件。一个使用插件的简单的 Win32 应用程序。这个应用程序展示了当前载入的插件，插件本身会在主程序中增加一些菜单项。插件可以被动态的加载和卸载。
- 第 12 章：运行期类型信息系统（RTTI）。一个自定义的运行期类型信息系统。它主要用于单继承。
- 第 12 章：多继承的 RTTI 系统。这是对前面自定义实时信息系统的改进。它通过添加一些附加信息来支持多继承。
- 第 13 章：对象工厂。这是一个模板化的游戏实体对象工厂。
- 第 14 章：序列化。一个非常简单的序列化程序，用于将完整的游戏实体树保存到磁盘中或从磁盘中载入。它使用一个自定义的流类和指针恢复方法。

所有的程序都使用 Visual Studio C++ 6.0 编译并在 Windows 2000 下进行运行。但插件程序实例是个例外，它是独立于特定平台和编译器的，所以，它们很容易在你所喜欢的环境下编译和运行。

提供了插件程序实例的可执行程序，不用经过编译就可以直接执行它。

关于附带光盘

本书讲述了一些复杂的概念，本书所带光盘中包含了演示这些复杂概念的程序的源代码。所有的代码都尽可能地简单，以便让读者能够抓住问题的关键；但与此同时，它们又是一些非常实用的程序，可以反映不同方法之间的联系以及它们是怎样整合在一起的。

本书并没有将一页页大量繁琐的代码都放进来，各章节的内容中只包含一部分比较重要的代码段。其余的细节可以参考光盘中的源代码。如果喜欢读大量的源代码，可以打开源代码文件，一边学习本书，一边领悟代码。

使用源文件的最好方法是编译文件来验证它的运行结果，如果想检查某种操作的执行过程可以使用单步调试方式，另外也可以对它们进行修改来看看如何扩展和改进它们。

每个程序的源文件和工程文件都放在一个单独的文件夹中。

- 第 7 章：内存管理。一个完整的内存管理器的完整实现和一个小型的测试程序。
- 第 11 章：插件。一个使用插件的简单的 Win32 应用程序。这个应用程序展示了当前载入的插件，插件本身会在主程序中增加一些菜单项。插件可以被动态的加载和卸载。
- 第 12 章：运行期类型信息系统（RTTI）。一个自定义的运行期类型信息系统。它主要用于单继承。
- 第 12 章：多继承的 RTTI 系统。这是对前面自定义实时信息系统的改进。它通过添加一些附加信息来支持多继承。
- 第 13 章：对象工厂。这是一个模板化的游戏实体对象工厂。
- 第 14 章：序列化。一个非常简单的序列化程序，用于将完整的游戏实体树保存到磁盘中或从磁盘中载入。它使用一个自定义的流类和指针恢复方法。

所有的程序都使用 Visual Studio C++ 6.0 编译并在 Windows 2000 下进行运行。但插件程序实例是个例外，它是独立于特定平台和编译器的，所以，它们很容易在你所喜欢的环境下编译和运行。

提供了插件程序实例的可执行程序，不用经过编译就可以直接执行它。